# Verification of Asynchronously Communicating Objects

Doctoral Dissertation by

Crystal Chang Din

Submitted to the
Faculty of Mathematics and Natural Sciences at the University of Oslo
for the degree Philosophiae Doctor in Computer Science


*Date of submission:* 03.03.2014
*Date of public defense:* 27.05.2014

Research group for Precise Modeling and Analysis
Department of Informatics
University of Oslo
Norway

Oslo, March 2014

*Series of dissertations submitted to the*
*Faculty of Mathematics and Natural Sciences, University of Oslo*
*No. 1494*

ISSN 1501-7710



Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the thesis defence. Kindly direct all inquiries regarding the thesis to the copyright holder or the unit which grants the doctorate.

# Abstract

The complexity and scale of software systems have dramatically increased in the past decades. Accordingly, it is becoming critical to reduce the effort in developing and maintaining software systems. Object orientation is a programming paradigm for modular development, in which programs can be reused and extended. Consequently, the amount of duplicated code may be decreased, and it is possible to expand the system in a systematical way.

Formal verification can be used to improve the quality of the software programs that are developed. Users may choose a specification language to formulate the properties of the program behavior in a precise manner and verify the program with respect to the specification within a formal reasoning framework.

Since most of the software systems are constantly modified and extended according to the system requirements or user requirements, it would be beneficial to reuse the unchanged modules by relying on their already proved specifications without knowing their actual implementation. The global properties of the whole system may be composed of the specifications of each module. Therefore, modularity should be reflected in software development as well as by the reasoning system.

The behavior of a software system can be captured by histories representing the communication between modules. In this thesis, concurrent and distributed settings are taken into account such that different modules can execute at the same time and possibly on different machines. In this setting, a modular reasoning system of an object-oriented language supporting concurrent objects is developed based on the use of communication histories, and it is proved sound and relatively complete.

# Acknowledgments

First and foremost, I appreciate the opportunity to pursue my PhD in the Precise Modeling and Analysis research group at the Department of Informatics at the University of Oslo. I would like to thank my main supervisor Olaf Owe for his excellent guidance, interesting discussions and great cooperation. I want to give thanks to my co-supervisor Einar Broch Johnsen for his clear intuition and critical suggestions. I am thankful for having Johan Dovland as another co-supervisor of mine. He guided me into my research field at the beginning.

I wish to thank Richard Bubel for rewarding collaboration. I enjoyed the time working together with him first at Chalmers University of Technology and later on at Technische Universität Darmstadt. I would like to thank Martin Steffen for his varied support over the years. My thanks also go to all my colleagues and students. They not only inspired me in the academic field but also made my life even more colorful. In particular, Silvia Lizeth Tapia Tarifa and me shared the same office for three years. We had lots of fun together along this PhD journey.

I would like to thank my dear families for their constant support, patience, love and prayers. I cherished my sister's visit during my PhD period. She attended the first lecture I gave in my life here in Norway. Finally, I would like to praise my Lord as He loves me and I am known by Him. He is always companying with me and leading my way.

# Contents

# Part I

# Overview

CHAPTER 1

# Introduction

Software creates new possibilities for our daily life. Some decades ago, transportation tickets were only sold on board. Now by the invention of online systems, we can purchase tickets at home. Software can replace human effort and do complex computation which cannot be achieved manually, for example, real-time weather forecast. Software can be embedded in numerous varieties of machines such as dish washers, air conditioners, and cars. Embedded systems hide the complex implementation but provide interfaces for the ease of use. Software can however cause crisis as well. A bug in the code controlling the Therac-25 radiation therapy machine was directly responsible for human death in the 1980s when the patients were exposed to excessive quantities of X-rays [77]. Therefore, a software with quality guarantee would not only increase the pleasure of the users but prevent the execution of software from loosing money or even human lives.

Compared to sequential programming which involves a consecutive and ordered execution, concurrent programming [9] increases the computation efficiency by allowing several computations to execute simultaneously and interact with one another. The execution of software is not restricted to stand-alone machines but can have cooperative and concurrent computations across the internet connection, which forms distributed systems. Consequently, concurrent programming can be applied on one or more than one computers. The actor model [4, 59] was introduced for concurrent computation. Actors communicate with one another by asynchronous message passing, which allows the caller to continue with its own activity without blocking while waiting for the reply. Each actor has its own virtual processor. An actor can send a finite set of messages to other actors, create a finite set of new actors and designate the behavior to be used for the next message it receives. Actions of different actors can be carried out in parallel in an actor system.

Object-oriented programming languages belong to a well-established paradigm for modular software development of concurrent and distributed systems [65]. Each code module contains everything necessary to execute only one aspect of the desired functionality. A modular software system can be constructed by combining already existing modules or extending it with newly built ones. Reusable code modules reduce the time spent producing duplicated code as well as provide flexibility by placing the same mod-

ule in different contexts. Concurrent objects combine object-orientation with the actor model. Each concurrent object has its own virtual processor and at most one process is active on an object at a time. In this setting, method executions are decoupled from method invocations using underlying asynchronous message passing. Concurrent objects communicating by such *asynchronous method calls* combine object-orientation and distribution in a natural manner, and therefore appears as a promising paradigm for distributed systems [27].

In general, a *future* [14, 53, 79, 97] is a data structure or can be seen as a placeholder that is created for a result that does not yet exist. The result of the future is computed concurrently with other executions and can be later collected. When combing the notion of futures with method invocations in the object-oriented setting, a fresh future identity is generated by the caller object upon invoking a method and the future is a placeholder for the method result. Future identities can be shared between objects, i.e., first-class futures. Through sharing future identities, the caller allows other objects to wait for the same method result or even delegates fetching of the method result to other objects. Compared to classical method calls, by which the callee returns the method result directly back to the caller, the use of futures results in faster, asynchronous, and more flexible computation.

The ABS language [52, 67] supports actor-based concurrency, object-orientation and first-class futures. ABS is a novel language for modeling object-oriented and distributed systems at an abstract, yet precise level. It is also a fully executable language and has code generators for Java [51], Scala [85], Maude [22], and Erlang [13]. By strong encapsulation, the internal state of each concurrent object is hidden from the environment, external access to the inner state of other objects is not allowed, neither at the programming level nor at the specification level.

Reasoning about concurrent systems is generally difficult, because of the need to consider all possible interactions among concurrently executing components. For distributed systems, reasoning about a component is also challenging since the environment of a component is usually unknown. The observable behavior of a system component interacting with the environment in a distributed setting can be captured in communication histories [7, 18, 23, 37, 44, 45, 63, 93, 94]. Accordingly, a possible approach to reason about concurrent and distributed systems is based on communication histories reflecting the interactions among system components. In addition, to improve verification efficiency, modularity in software development should be reflected at the reasoning level in terms of verifying each module and composing the result at the end. ABS is a successor of the Creol language [69]. However, Creol does not support first-class futures. The languages we consider in this thesis are inspired by Creol/ABS. In [7, 44, 45], history-based compositional reasoning systems for Creol have been developed.

For convenience, in Part 1 of this thesis we will refer to the setting of concurrent objects communicating by asynchronous method calls and first-class futures by the term *concurrent asynchronous objects*.

## 1.1 Research Goals

The verification of concurrent and distributed systems is generally complicated and difficult to achieve. Therefore, the overall goal of this thesis is to achieve effective reasoning, i.e.,

> *to simplify reasoning about concurrent objects communicating*
> *by asynchronous method calls using history-based invariants*
> *and to provide corresponding tool support.*

To achieve this goal, this thesis will address the following questions:

1. How can we develop a history-based reasoning system for concurrent asynchronous objects which is simpler than the existing one for Creol?
2. As mentioned above, futures improve the communication efficiency between objects. However, futures are shared entities between objects. How can we achieve local reasoning inside each class when dealing with futures?
3. How can we prove the soundness and completeness of our reasoning system?
4. How can we realize our reasoning system by means of executable tools?

## 1.2 Outline

Chapter 2 reviews some main characteristics of concurrency, Chapter 3 describes general principles of object-oriented languages and introduces the ABS language, Chapter 4 considers reasoning in the object-oriented setting, and Chapter 5 gives a short summary of each paper. Chapter 6 gives an evaluation of the contribution of this thesis towards the research goals stated in this chapter and briefly discusses about some possible future work.

CHAPTER 2

# Concurrency

To explain the concept of concurrency, I would like to start with a short introduction of sequential programs. In sequential programs, the syntax determines the order. Typically, the execution sequence is deterministic. Given the same input, the program always provides the same output. In concurrent programming, each process (or thread) is in charge of a sequential execution of statements. There are two possible approaches for executing the processes. The first approach is by assigning more than one process to a single processor but at most one process is executed by the processor at a time. The operating system scheduler decides which process to be executed and at what time. Compared to sequential programming, processes in this setting may interleave with one another on the same processor in the sense that the execution of one process may be interrupted by the execution of another process and continues the execution later according to the scheduler. Since the time slide for different process executions is extremely short, the users perceive the tasks as running at the same time. The second approach for executing processes is by assigning processes to different processors such that the processes may actually be executed in parallel.

In general, we can distinguish the following three interaction models for concurrent processes: shared variables, remote method invocation, and message passing [9]. Accordingly, processes cooperate with one another to accomplish a common task or solve a problem. The description and the comparison between them are illustrated in the following subsections.

## 2.1 Shared Variable Communication

By using shared variable communication, a process writes a value to a memory space where another object may later read the value. Synchronization is required here in order to obtain a correct runtime order of the shared-variable access and avoid race conditions [83].

Let's look at an example for making withdrawals from a checking account represented by the shared resource *balance*:

```
1. bool withdraw( int amount ){
2.   if ( balance >= amount ){
3.         balance = balance - amount;
4.         return true;
5.   }
6.   return false;
7. }
```

Suppose *balance* equals 300, and two concurrent processes make the calls $withdraw(200)$ and $withdraw(150)$. If line 3 in both operations executes before line 5, both operations will find that $balance >= amount$ evaluates to true, and execution will proceed to subtracting the *amount*. However, since both processes perform their withdrawals, the total amount withdrawn will end up being more than the original *balance*. These sorts of problems with shared resources require the use of concurrency control ensuring that correct results for concurrent operations are generated.

One of the synchronization techniques for concurrency control is to protect the critical sections, where reading or writing actions to the shared variables are performed, such that the execution of this block can be seen as an atomic action and at most one process can execute in this block at a time, i.e., mutual exclusion is guaranteed. Locks, semaphores [33], and monitors [62], are synchronization constructs to protect the access to the mutable shared resources within a critical section.

Another synchronization technique is called conditional synchronization. A process needs to wait for a condition to be satisfied in order to proceed with the execution. This technique can be used when a process needs to wait until another process has finished a certain task and then proceeds with the execution, or all the processes need to wait for one another until all have reached a certain program point and then proceed to the next execution section together.

Since synchronization limits the possible executions between processes, if we represent an execution of the concurrent system as a communication history between processes, synchronization is reflected by the reduced number of possible communication histories.

**Cache Coherence Problems.** In hardware systems, caches are used for temporary storage of data likely to be used again. Each thread may have its own cache which stores some copies of data from the memory such that the thread does not need to go to memory for every reference to a variable. Accordingly, the data stored in the cache could be easily fetched and reused. However, by combining shared memory communication with the use of caches, memory consistency problems may occur. This happens when one thread modifies its copy of the data, the other thread will then have a stale copy of the data in its cache. Cache coherence [12] is intended to manage such conflicts and maintain consistency between cache and memory.

## 2.2 Distributed Programming

In the shared memory model, the processes usually communicate with one another by reading and writing shared variables on the same machine. However, processes can also be allocated to different processors located on different machines. In this case each processor has its own specific memory. Consequently, shared variable concurrency is not the most suitable way of communication between processes in loosely coupled systems where communication takes time and inherent latency and unreliability exist in the network. In this setting, communication between the processes is commonly through a network by means of sending remote method invocations or exchanging the messages with one another.

With remote method invocations, the thread of control is transferred with the call and the caller activity is blocked until the return values from the call have been received, as in the case of Java RMI [5]. This blocking of internal activity makes it difficult to combine active behavior in an object with the processing of requests from the environment. In a distributed setting, synchronous communication therefore gives rise to undesired and uncontrolled waiting, and to possible deadlocks.

A method call can also be formed by an invocation and a reply message through various mechanisms, including channels. Compared to remote method invocation, message passing does not transfer the execution control between the communicating units and may be synchronous or asynchronous. By synchronous communication, both sender and receiver must be ready before communication can occur, as in Ada's rendezvous mechanism [64] or Hoare's CSP [63]. Hence, the objects synchronize on the message transmission. Remote method invocation may be captured in this model if the calling object blocks between the two synchronized messages representing the call [9]. By asynchronous communication, the participants need not be ready for each other at the same time. The caller's activity is synchronized with the arrival of the reply message rather than with the emission of the invocation. The message-receiving order may then be different from the message-sending order. Unnecessary waiting between objects can be avoided. Therefore, asynchronous communication provides better efficiency over distributed systems. Examples include e-mail, discussion boards and text messaging over cell phones.

**The Actor Model.** The actor model [4, 59] was introduced in 1973 for concurrent computation. An actor is a computational entity. Each actor encapsulates its fields and a single thread of control. Actors communicate asynchronously with each other through message passing. Accordingly, actors do not block while waiting for the responses to their messages. Since there is no shared state between actors, actors never need to compete for locks in order to access the shared data. The actor model is represented by languages such as Erlang [13], Scala [85], Creol [69], and ABS [52, 67].

CHAPTER 3

# Object Orientation

Object orientation is the leading framework for concurrent and distributed systems. It has been claimed that object orientation and distributed systems form a natural match and is recommended by the RM-ODP [27, 65]. Simula (1967) [26] is generally accepted as the first language that introduced the primary features of object-orientation.

Two central concepts in object-oriented programming are classes and objects. A well-designed class typically consists of data declarations that represent the fields of the class, and behavior representation as a set of methods. Objects are generated at run-time as instances of classes, and each object has a unique identity. The state of the object is formed by the values of its fields, and the behavior of the object is represented by its methods. The collection of classes forms the static view of a program. The dynamic view of the application is a set of objects executing and interacting with one another.

Due to the language support for modular programming style, code reuse, as well as encapsulation of individual objects, object-oriented programming languages are widely used in software development. In this setting, software systems are built by adapting and combing already existing code modules without knowing their implementation details.

Some other object-oriented languages are, for example, BETA [73], Smalltalk [50], Modula-3 [20], C++ [95], Eiffel [81], Java [51], C♯ [58], Scala [85], and the modeling language Creol [69]. ABS [52, 67], which we will talk about in the end of the chapter, is a successor of Creol.

## 3.1 Modularity

Modular programming is a technique that keeps the complexity of a large program manageable by systematically splitting the program into different components, i.e., modules. Accordingly, a problem can be divided into subproblems, and the solution to a problem consists of several smaller solutions corresponding to each of the subproblems. Object-oriented programming enhances modular design by providing classes as the basic modular unit. A program structure is basically organized as a collection of

classes.

To write a new module which relies on the functionalities of other modules, the programmer has an assumption of what the other modules provide, i.e., the specifications of other modules, and not how those modules are implemented. Accordingly, several programmers can work on separated modules at the same time, thus making development of program faster, and the code base is easier to debug, update and modify.

## 3.2 Encapsulation and Information Hiding

The first step towards modularity in object-oriented programming is through encapsulation, such that the internal representation of an module is hidden from the environment. Only the relevant aspects of the module are exposed. An abstract data type (ADT) [78], an interface, or even up to a system can all be seen as a module.

An ADT is a combination of a data structure with a set of operations on the contained data. For example, one may choose to store the data in an array, in a linked-list, or in some other data structure. The operations on the contained data may be to add data to a data structure, remove data from a data structure and search data from a data structure. The implementation decision for the operations may then depend on the data structure and vary from application to application. The users of an ADT do not need to know how the ADT is implemented but only the functionality that the ADT provides.

Many object-oriented languages support the notion of interfaces, which declare a set of method signatures for different operations but do not provide the implementation details of the methods. A class that implements an interface (or more than one interface) must implement all of the methods described in the interface. An object is an instance of a class and is typed by the interfaces that the class implements. The internal representation of an object is generally hidden from view outside of the object's definition and via keywords like *public*, *protected*, and *private*, the programmers have a degree of control over how much an object can be exposed to the environment. By strong encapsulation, only the object's own methods can directly inspect or manipulate its fields, such that the internal data are protected from being set into invalid or inconsistent state by the surrounding environment.

## 3.3 Inheritance

In order to build new software in a modular way, the existing software is required to be reusable. Inheritance was invented in 1967 for Simula [26] as a mechanism for code reuse. Inheritance allows classes to extend fields and methods from existing classes, and to add extra attributes and methods to its own implementation. The resulting classes are known as derived classes or subclasses, and the resulting hierarchy is known as a class hierarchy. In single inheritance, a class may extend at most one class. This is distinct from multiple inheritance, where a class may extend the characteristics

and features of more than one classes. Multiple inheritance does provide flexibility in reusing the code from different classes. However, it also increases complexity in case of conflicting inherited features. For instance, the "diamond problem" [72] may cause ambiguity as to which parent class a particular feature is inherited from if more than one parent classes implement said feature. Inheritance is not considered in this thesis work. We refer to [46] for a treatment of inheritance.

## 3.4 Object Interaction

Two basic mechanisms for object interaction are by remote field access or method invocation. Remote field access allows an object to operate directly on the fields of other objects, usually by dot notation. For example, an object $o$ may store the value of the field $x$ of object $r$ in its variable $v$ by executing $v := r.x$, or object $o$ may modify the value of $x$ of $r$ by $r.x := r.x + 5$. On the other hand, object interaction by only method invocations restricts the modification of the remote fields to be manipulated only at the callee side. For example, an object $o$ may increase the value of the field $x$ of object $r$ by invoking a method $add(int\ i)\{x := x + i;\}$ on $r$, such that the execution of $r.add(5)$ increases $x$ by 5.

## 3.5 The ABS Language

In thread-based languages such as Java, the execution threads are separated from objects. Several threads may operate simultaneously on the same object. The encapsulation of object orientation is broken. This often leads to a low-level style of programming based on, e.g., the explicit manipulation of locks. Safety is by convention rather than by language design [54]. However, this obstacle is avoided in the actor-based approach by encapsulating control within actors.

The Abstract Behavioral Specification language (ABS) [52, 67] targets distributed and object-oriented systems and permits actor-style asynchronous communication [59]. ABS is a fully executable language and has code generators for Java [51], Scala [85], Maude [22] and Erlang [13]. ABS abstracts away many implementation details which are not desirable at the modeling level such as I/O implementations, and the concrete representation of internal data structures. ADTs are supported by ABS, as well.

ABS supports first-class futures, i.e., futures [14, 53, 79, 97] which can be shared between objects. A fresh future identity is generated by the caller upon invoking a method. The caller does not wait for the return value after invoking the method but continues its computation. We say futures are resolved when the callee stores the method result in the future upon termination of the method execution. After the futures are resolved, they become read-only. By sharing future identities, the caller allows other objects to wait for the same method result from the same future, and may even delegate fetching of the method result to other objects. After the future becomes resolved, all the objects which are the future-identity holders are free to fetch the value from the future as many times as they require and in any order. Compared

to classical method calls, by which the callee returns the method result directly back to the caller, futures offer greater flexibility in application design and can significantly improve concurrency in object-oriented paradigms.

In ABS, internal interference of an object is avoided since each object has its own virtual processor and at most one process is executing on an object at the time. Nevertheless, deadlock may occur in the ABS setting when, for instance, the callee sends a method call to the caller while the caller is blocking and waiting for a future value, from the callee. Fortunately, the notion of *process release point* is supported in the ABS language. A process is released at a process release point, expressed using a Boolean guard, when a boolean condition is evaluated to false or a future being queried is not resolved yet. While the current process is released, other available processes can be chosen by the scheduler for execution. Therefore, the use of process release points influences the control flow inside concurrent objects by providing cooperative scheduling of the method activities. Accordingly, the blocking problem mentioned above may be overcome by using the combination of asynchronous method calls and process release points. The method call from the callee object will be executed on the caller object as soon as any ongoing method execution of the caller object reaches the end or a release point.

A method call in ABS does not transfer the execution control from the caller to the callee and a method call leads to a new process on the called object. Object interaction is only by asynchronous method calls. Remote field access is not supported by the language, so there is no shared variable communication between different objects. Consequently, concurrent objects are disjoint such that the processing steps made by one object do not affect the other objects. The concurrent object model of ABS is inherently compositional.

The languages we consider in this thesis are inspired by ABS and ignore language features that are orthogonal to concurrent asynchronous objects, such as component object groups and the language-based support for product line engineering [89].

CHAPTER 4

# Program Analysis for Object-Oriented Concurrent Systems

To build reliable software systems, it is important to develop techniques which facilitate reasoning about the behavior of the program code. We organize the discussion in this chapter by first considering programming logic, then discussing abstraction mechanisms in program analysis, after that looking more specifically into the challenges of reasoning about object-oriented concurrent systems. Finally, we will present the reasoning system we develop for concurrent asynchronous objects.

## 4.1 Programming Logic

A formal reasoning system provides a structural and mathematical way to verify the behavior of program code. Hoare logic [60] describes how the execution of a piece of code changes the state of the computation by an integration of programs and assertions within a single framework. A "Hoare triple" expressing *partial correctness*, i.e., program termination is not concerned, is of the form $\{P\}\ S\ \{Q\}$, which has the following meaning:

> If the execution of a program $S$ starts in a state where the assertion $P$ holds and the program terminates normally, the assertion $Q$ will hold in the final state.

$P$ and $Q$ are the pre- and post-condition of the program $S$, respectively. Notice that if the program does not terminate normally due to infinite loops or the program aborts because some error situations have occurred, the postcondition of the Hoare triple does not need to hold. Accordingly, any precondition $P$ for a program $S$ such that $\{P\}\ S\ \{\mathit{false}\}$ holds, is a sufficient condition for abnormal behavior of $S$.

Assertions are defined over the program variables. *Auxiliary variables* do not appear in the program code and they are used in the assertions to relate the values of

the program variables in different states [71]. For example, the auxiliary variable $x_0$ in the Hoare triple $\{n = x_0\}n := n + 5\{n = x_0 + 5\}$ is used to store the value of the program variable $n$ in the prestate such that the initial value of $n$ may be referred to in the postcondition. The meaning of $\{P\}\ S\ \{Q\}$ with auxiliary variables is as before but must hold for all values of the auxiliary variables.

A total correctness specification, in which program termination is required, is expressible in a strengthened Hoare logic. The Hoare triple of the form $[P]\ S\ [Q]$ has the following meaning:

> If the execution of a program $S$ starts in a state where the assertion $P$ holds, the program will terminate and the assertion $Q$ will hold in the final state.

In this thesis, we will restrict ourselves to partial correctness.

### 4.1.1 Formal Proofs

A proof is a sequence of reasoning steps designed in order to convince the reader about the truth of some formulae, i.e., a theorem. In order to do this the proof must lead from axioms which are obviously true to the theorem, by applying proof rules (which are also called inference rules or deduction rules).

A proof may be seen as a tree with axioms as leaves and the main theorem as the root. Each internal node of the proof tree is a consequence of its immediate descendant nodes according to given proof rules. For example, a proof rule for sequential composition may be formulated in the following format using Hoare triples:

$$\frac{\overbrace{\{P\}S_1\{M\} \qquad \{M\}S_2\{Q\}}^{Premisses}}{\underbrace{\{P\}S_1; S_2\{Q\}}_{Conclusion}}$$

This proof rule expresses that we may prove a sequential composition $S_1; S_2$ with a precondition $P$ and a postcondition $Q$, if we can find an intermediate condition $M$ such that both $\{P\}S_1\{M\}$ and $\{M\}S_2\{Q\}$ are provable. A formula is provable if we can continue applying proof rules until each branch of the proof tree is closed with an axiom. Accordingly, for proving the *Conclusion*, it suffices to prove all *Premisses*. An example of an axiom is $\{P\}S\{true\}$, which is accepted as true without controversy.

### 4.1.2 Gentzen-style Sequent Calculus

Sequent calculus [49] was introduced by Gentzen. A sequent is in the form of

$$\overbrace{\psi_1, \ldots, \psi_m}^{antecedent} \vdash \overbrace{\phi_1, \ldots, \phi_n}^{succedent}$$

where the antecedent formulae $\psi_i$ are the assumption part of the sequent and the succedent formulae $\phi_i$ are the theorem part of the sequent. The interpretation of a

sequent is that the truth of the theorem part $\phi_1 \vee \ldots \vee \phi_n$ follows from the truth of the assumption $\psi_1 \wedge \ldots \wedge \psi_n$. A sequent calculus rule is in the form of

$$\frac{Seq_1;\ Seq_2;\ \ldots;\ Seq_n}{Seq}$$

Given sequents $Seq_1$, $Seq_2$, ..., $Seq_n$ (n $\geq$ 0), for premises, we may infer a sequent of the form $Seq$, the conclusion.

### 4.1.3 Dynamic Logic

Dynamic logic [55, 90] is another example of program logic. It was developed by Vaughan Pratt and can be seen as an extension of Hoare logic. The dynamic logic formula $P \rightarrow [S]Q$ is similar to the Hoare triple $\{P\}\ S\ \{Q\}$ expressing partial correctness, and the dynamic logic formula $P \rightarrow \langle S\rangle Q$ is similar to the Hoare triple $[P]\ S\ [Q]$ expressing total correctness. However, the assertions $P$ and $Q$ are pure first-order formulae in Hoare logic, whereas they can contain programs in dynamic logic. In dynamic logic we are able to use a program to specify that a data structure is acyclic, which is not possible in pure first-order logic. Therefore, dynamic logic is more expressive than Hoare logic.

We may use Gentzen-style sequent calculus to prove dynamic logic formulae. A sequent may reformulate $P \rightarrow [S]Q$ into

$$\Gamma, P \vdash [S]Q, \Delta,$$

where $\Gamma$ and $\Delta$ stand for (possibly empty) sets of formulae. Below is the proof rule for the if-else statement:

$$\frac{\Gamma, b \vdash [\texttt{p};\texttt{rest}]\phi, \Delta \qquad \Gamma, \neg b \vdash [\texttt{q};\texttt{rest}]\phi, \Delta}{\Gamma \vdash [\texttt{if(b)\{p\}else\{q\}};\texttt{rest}]\phi, \Delta}$$

The application of this proof rule splits the proof into two branches. The left branch assumes that the guard of the conditional statement is true. Here, we have to show that after execution of the *then* branch of the conditional and the rest of the program, we are in a state in which formula $\phi$ holds. The right branch is concerned with the analogue case where the guard is assumed to be false.

### 4.1.4 Soundness and Completeness

A formula expressed in a formal language is *valid* if and only if it is true under every interpretation, i.e., an assignment of meaning to the symbols of the formal language. For programming languages, *semantics* describes the behavior that a computer follows when executing a program in the language. A deductive system is said to be *sound* with respect to a semantics if all provable formulae are valid. A deductive system is said to be *complete* with respect to a semantics if all valid formulae are provable.

## 4.2 Abstraction in Program Analysis

Specification languages generally rely on abstractions of one kind or another since specifications are typically defined at a more abstract level than the actual implementation.

**Rely/Guarantee Reasoning.** In modular reasoning, each module is specified and verified independently from its environment. However, the behavior of each module may be affected by its environment. In rely/guarantee reasoning [31, 70], the specification of a method is a quadruple $(p,R,G,q)$, where $p$ and $q$ are pre- and post-conditions, $R$ and $G$ are rely- and guarantee-conditions. A method satisfies its specification if, given that $p$ is satisfied in the initial state and given an environment whose behaviors satisfy $R$, each atomic transition made by the execution of the method satisfies $G$ and $q$ is satisfied in the final state. We can see rely-conditions as an abstraction of the environment and guarantee-conditions as an abstraction of what the method promises to provide.

**Class Invariants.** In object-oriented programs, classes are the basic modules. A *class invariant* provides a way of abstracting the state of an object in one formula and allows you to have reasoning knowledge of an object without knowing its exact state. Accordingly, a class invariant of a class $C$ specifies invariant properties of instances of $C$.

**Model Fields.** We may specify class invariants by using *model fields* [21, 74] without necessarily exposing the class implementation so that information hiding can be supported. Model fields are specification-only fields and cannot be used in the program code. The values of the model fields are determined by the concrete fields. Therefore, a model field is an abstraction of the state. For example, a class *Account* contains two fields, *income* and *expense*. A model field in this case can be a variable *saving* defined as the following:

$$saving \triangleq income \text{ - } expense$$

Therefore, a model field can be seen as the abstraction of the concrete fields [21], and the values of the model fields determines its abstract value [61]. A class invariant for *Account* using the model field *saving* can be

$$saving > 5000. \tag{4.1}$$

**Communication Histories and Ghost Fields.** Another way to achieve abstraction in program analysis is by writing specifications in terms of potential observable behavior. In object-oriented programs, the observable behavior is the communication between the objects, which can be captured in the *communication histories* as a sequence of observable *communication events* between objects [7, 18, 23, 37, 44, 45, 63, 93, 94]. We can define functions on the communication histories, i.e., *history functions*, to

extract information from the histories. Refer to the *Account* example above, we can define a history function, *Savings(H)*, which extracts the values of *income* and *expense* from the communication history $H$. Notice that we may relate model fields to history functions such as a history-based class invariant for *Account* can be reformulated from (4.1) into

$$Savings(H) > 5000.$$

Notice that the communication history is more expressive than model fields. We may be able to relate model fields to history functions but not the other way around. If histories are too complex then it is not ideal to use histories in the specifications, we can replace history functions with model fields such as

$$saving = Savings(H).$$

A *ghost variable* [74] (or a mythical variable in [25]) is similar to a model field and can only be used for specifications. Unlike a model field, a ghost variable is not a pure abstraction of the state. It is an extension of the semantics and does not have a value determined by the concrete fields; instead its value is directly determined by its initialization or by a set-statement. Therefore, a ghost variable is different from auxiliary variables in a sense that the value of the ghost variable is modifiable. We may see communication histories as ghost variables.

## 4.3 Reasoning Challenges for Object-Oriented Concurrent Programs

The basic problem for program analysis of programs with concurrency is discussed in [9]. In this section, we will first point out some reasoning challenges for object-oriented concurrent programs with shared variable concurrency. After that we will discuss the challenges for reasoning about multithreaded systems. At the end, we will present the reasoning approach we develop in this thesis and the advantage of using it to reason about concurrent asynchronous objects.

### 4.3.1 Shared Variable Communication

Remote access to the internal state variables by dot-notation complicates reasoning. In the following, we will discuss some related issues.

**Remote Access and Aliasing.** If an object $o$ can directly modify an object $r$'s fields, e.g., $r.a := 5$, the satisfaction of the class invariant of $r$ needs to rely on $o$. On the other hand, if the invariant of the object $o$ depends on the fields of $r$, it must be proved that the methods in $r$ maintain the class invariant of $o$. Therefore, modularity is broken. The situation is further complicated when remote access is combined with aliasing. This is because the static reasoning needs to account for the aliasing which may exist during the execution. Aliasing occurs when more than one variable refers

to the same object. Therefore, if both variables $v_1$ and $v_2$ refer to the same object $o$, i.e., $v_1 := o; v_2 := v_1$, the modification of $v_1$ on $o$'s fields should be transparent to the variable $v_2$.

**Reasoning about Shared Variable Communication using Model Fields.** An update of a class field has an instant effect on all the model fields it relates to, therefore modularity may be broken in a setting with shared variable concurrency. This problem can be illustrated by the following example: a class $C_2$ is implemented after a class $C_1$, and $C_2$ has a field of class $C_1$. If the value of a model field $m_f$ of $C_2$ depends on a model field of $C_1$, an update of the fields of $C_1$ may simultaneously change the value of $m_f$ of $C_2$. Since the implementor of $C_1$ need not be aware of $C_2$, this case leads to a modularity problem and causes a potential invariant violation for $C_2$.

A solution which builds on the Boogie methodology is provided in [15, 76]. In the Boogie methodology, an object is either in a valid or a mutable state. The transition from valid to mutable and back is performed by two special statements, **unpack** and **pack**. An object is guaranteed to satisfy its invariant only when in a valid state, and only fields of objects in a mutable state can be assigned. Model fields are updated only by a special **pack** statement. Consequently, the updates of the concrete state do not automatically change the values of model fields. This methodology guarantees that whenever an object is manipulated, the invariant of the owner cannot be assumed to hold, and the specified relation between a model field and the concrete state of an object holds whenever the object satisfies its invariant in a valid state. However, one would need more fine-grained control of when a class invariant holds.

### 4.3.2 Multithreaded Systems

In multithreaded object-oriented systems with shared variable concurrency, each object encapsulates and protects a set of shared variables and may contain several threads. However, multiple threads can interfere with each other when accessing the shared variables in the same object. Consequently, the synchronization techniques such as locks, semaphores [33] and monitors [62] need to be carefully used in the implementation of classes. Verification of multithreaded systems can be found in [1, 75]. Below we will first point out some challenges of rely/guarantee reasoning in this program setting as well as some approaches to handle the related problems. Then we will discuss a synchronization technique, monitors, in multithreaded systems.

**Rely/Guarantee Reasoning for Multithreaded Systems.** When we reason about multithreaded systems with shared variable concurrency using rely/guarantee techniques, the whole program state is viewed as a shared resource, and each thread views the set of other threads in the system as its environment. In order to ensure non-interference between the thread and its environment, the following property needs to be satisfied: If the current state satisfies the precondition $p$ but the current thread is preempted by its environment, p still holds and the environment satisfies its rely condition $R$ when the current thread resumes its execution in a new state. Notice that

$R$ needs to capture all possible behaviors of the environment and $G$ needs to specify threads' behaviors over the whole object state. Therefore, this requirement makes it difficult to define $R$ (and $G$).

In addition, rely/guarantee reasoning related to modularity has the following challenges. The thread-private resource has to be exposed in the specifications even if a part of the state might be locally owned by a single thread. Similarly, all the resources need to be exposed to all the threads even if some of them might be shared only by a subset of the threads. Even if some components of the system only access a part of the shared resources, the rely/guarantee conditions of the components still need to satisfy all the shared resources. Due to all these challenges, it is hard to achieve compositional reasoning in the rely/guarantee approach with shared variable concurrency. Some solutions are proposed in [48, 96] combing rely/guarantee reasoning with separation logic [91]. Accordingly, specifications and proofs of a program component mention only the portion of memory used by the component, and not the entire global state of the system.

**Monitors.** A monitor [62] can be defined as a thread-safe module that uses mutual exclusion in order to safely allow access to a method or variable by more than one thread. Shared data in the monitor is encapsulated and is manipulated using monitor methods. Only the method's name is visible from outside the monitor. Statements inside a monitor do not have access to variables outside the monitor. For synchronization, monitor supports implicit mutex. At most one thread can access the monitor method at a time. Therefore, freedom from interference for monitor execution is guaranteed, and the programmer does not need to use any ad hoc methods for mutex. The monitor's inner state can be described by a *monitor invariant*, which needs to be maintained by execution of methods. A monitor invariant must hold after initialization, when the thread is suspended and when a method execution terminates, assuming the invariant holds when a method execution starts and after a thread resumes the execution.

### 4.3.3 Program Analysis for Concurrent Asynchronous Objects

In our setting (as in Creol/ABS), concurrent objects can be seen as monitors with cooperative scheduling and asynchronous communication. Remote field access is not allowed and object interaction is only by asynchronous method calls, i.e., there is no direct execution control transferred from the caller to the callee. Therefore, aliasing does not break the modularity of reasoning. Internal interference in a concurrent asynchronous object is avoided since at most one process is active on an object at a time. Accordingly, program analysis for concurrent asynchronous objects follows the monitor approach and we may define an invariant for each class for *local reasoning* about classes.

The communication between the objects, i.e., the observable behavior, can be captured in the communication histories as a sequence of observable communication

events between objects [18, 63]. At any point in time the communication history abstractly captures the system state [24, 25]. In fact communication histories are used in semantics for full abstraction results (e.g., [2, 66]). In this thesis, the semantics of concurrent asynchronous objects inspired by ABS describes how the execution of each statement influences the extension of the local communication history of each object. Moreover, communication histories are expressible in our assertion language. The *class invariant* may express the potential interaction between the instances of the class and the environment in terms of the relation between the observable behavior and the changes of the internal state. Similar to a monitor invariant, the class invariant is required to hold after initialization in all the instances of the class, before any process release point and upon termination of the method executions. Consequently, whenever the process is released, either by the termination of a method execution or by a process release point, the process that gains the execution control can then rely on the class invariant.

The global history of the whole system is formed by the assembly of the local history of each instance of the class. However, object communication in concurrent asynchronous objects is by asynchronous method calls, so messages may in general be delayed in the network. The observable behavior of an object system allows the order of the messages received by the callee to be different from the order of the messages sent by the caller. The knowledge of message ordering in our setting is captured by a global notion of *wellformed history*.

The use of communication histories supports compositional reasoning. First, we ensure information hiding for each component, i.e., class. This is achieved by defining the *object invariant* which is expressed in terms of the class invariant by hiding the internal state of the object. Accordingly, the object invariant only exposes the abstract behavior of the class instances in terms of the local communication history. Since the global history forms the connection between object invariants, the specification of the global system may be derived by composing the specifications of each object under the assumption of history wellformedness. A *history invariant* is the specification of the global system which needs to hold for all finite sequences in the prefix-closure of the set of possible histories, expressing safety properties [8]. Therefore, object invariants should also be prefix-closed. If necessary, we may derive prefix-closed object invariants by weakening class invariants. *Global program reasoning* in the setting of concurrent asynchronous objects is done by verifying each class against the class invariant.

CHAPTER 5

# List of the Research Papers

This chapter gives a short summary of each research paper in Part II of this thesis. The contents of the papers appear as in their original publication, but have been reformatted to fit the layout of this thesis.

## 5.1 Paper 1

| | |
|---|---|
| **Title:** | Observable Behavior of Distributed Systems: Component Reasoning for Concurrent Objects |
| **Authors:** | Crystal Chang Din, Johan Dovland, Einar Broch Johnsen and Olaf Owe |
| **Publication:** | The Journal of Logic and Algebraic Programming (2012) [37] |

**Summary:** In this paper, an abstract and yet executable modeling language is considered. This language, based on concurrent objects communicating by asynchronous method calls, avoids some difficulties related to compositionality and aliasing. To facilitate system analysis, compositional verification systems are needed, which allow components to be analyzed independently of their environment. A proof system for partial correctness reasoning is established based on communication histories and class invariants. A particular feature of our approach is that the alphabets of different objects are completely disjoint. The soundness and relative completeness of this proof system are shown using a transformational approach from a sequential language with a non-deterministic assignment operator.

## 5.2 Paper 2

| | |
|---|---|
| **Title:** | A Sound and Complete Reasoning System for Asynchronous Communication with Shared Futures |
| **Authors:** | Crystal Chang Din and Olaf Owe |
| **Publication:** | The Journal of Logic and Algebraic Programming (to appear) |

**Summary:** This paper presents a Hoare style reasoning system for distributed objects based on a model for asynchronously communicating objects, where return values from method calls are handled by futures. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported, and each object may be specified and verified independently of its environment. The presented reasoning system is proven sound and relatively complete with respect to the given operational semantics.

## 5.3 Paper 3

| | |
|---|---|
| **Title:** | Compositional Reasoning about Active Objects with Shared Futures |
| **Authors:** | Crystal Chang Din and Olaf Owe |
| **Publication:** | Submitted to The Journal of Formal Aspects of Computing<br>A short version of this paper has been published at SEFM'12 [39] |

**Summary:** In this paper, a general concurrency and communication model focusing on asynchronous method calls and futures is presented. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported, as each object may be specified and verified independently of its environment. Soundness of the composition rule is proved. A kernel object-oriented language with futures inspired by the ABS modeling language is considered. A compositional proof system for this language is presented and formulated within dynamic logic such that the reasoning rules can be directly implemented in the KeY framework.

The languages in Papers 2 and 3 are slightly different. Paper 2 contains interfaces which Paper 3 does not, and Paper 3 contains while-loops which Paper 2 does not.

## 5.4 Paper 4

| | |
|---|---|
| **Title:** | A Comparison of Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems |
| **Authors:** | Crystal Chang Din, Olaf Owe and Richard Bubel |
| **Publication:** | Submitted to The Journal of Logic and Algebraic Programming<br>A short version of this paper has been published at MODELSWARD'14 [43] |

**Summary:** We investigate the usage of a history-based specification approach for concurrent and distributed systems by first implementing the reasoning systems provided by Papers 2 and 3. After that, we compare two approaches on checking that those systems behave according to their specification. Concretely, we apply runtime assertion checking and static deductive verification on two small case studies to detect

specification violations, respectively to ensure that the system follows its specifications. We evaluate and compare both approaches with respect to their scope and ease of application. We give recommendations on which approach is suitable for which purpose as well as the implied costs and benefits of each approach.

## 5.5 Further Papers

In the following, we list further papers which are technically not presented as part of this thesis, or correspond to shorter and preliminary versions of the work reported in this thesis.

**Related to Paper 1:**

- *Observable Behavior of Distributed Systems: Component Reasoning for Concurrent Objects* has been presented as an extended abstract at the Nordic Workshop on Programming Theory (NWPT'10) [36].
- A preliminary version has been published as UiO research report [35].

**Related to Paper 2:**

- *Soundness of a Reasoning System for Asynchronous Communication with Futures* has been presented as an extended abstract at the Nordic Workshop on Programming Theory (NWPT'12) [40].

**Related to Paper 3:**

- *Compositional Reasoning about Shared Futures* has been published at the International Conference on Software Engineering and Formal Method (SEFM'12) [39].
- A preliminary version has been published as UiO research report [38].

**Related to Paper 4:**

- *A Comparison of Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems* has been presented as an extended abstract at the Nordic Workshop on Programming Theory (NWPT'13) [34].
- A preliminary version has been published as UiO research report [41].
- *Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems* has been published at the International Conference on Model-Driven Engineering and Software Development (MODELSWARD'14) 2014 [43]

CHAPTER 6

# Discussion

In this chapter, we return to the research questions listed in Section 1.1 and we summarize the contributions of the thesis by answering the questions. In the end, we discuss the future work.

## 6.1 Summary of the Contributions

**1: How can we develop a history-based reasoning system for concurrent asynchronous objects which is simpler than the existing one for Creol?**

In Paper 1 we develop a four-event semantics for classical method communications, where the callee returns the method result directly back to the caller. A method call cycle is reflected by four different kinds of events, i.e., the first type of events capture method invocations, the second type of events capture the starting point of the method executions, the third type of events capture the termination of the method executions and the fourth type of events capture the fetching of the method results. In addition, object generation is reflected in the creation events. Each event is generated by one and only one object. Therefore, the disjointness of concurrent objects communicating by asynchronous method calls implies that the generated local histories are disjoint, in the sense that processing steps made by one object do not affect the history-based invariants of the surrounding objects.

The former history-based reasoning system for Creol is based on a two-event semantics for method calls [7, 44, 45], where events are visible to more than one object. For instance, message sending is visible on the local history of the receiver. The local histories must then be updated with the activity of other objects, resulting in more complex reasoning systems.

**2: Futures improve the communication efficiency between objects. However, futures are shared entities between objects. How can we achieve local reasoning inside each class when dealing with futures?**

Futures are global entities shared between objects. Local reasoning inside a class involving method communication using futures is therefore challenging. We solve this problem by adapting our four-event semantics for classical method communications from Paper 1. Each event is generated by one object. Disjointness is guaranteed at the language level between objects and at the reasoning level using histories. However, the difference is: all the communication events relating to method invocations, method executions and value fetching from futures contain a future identity. Accordingly, we can define history-based class invariants expressing future-related properties for local reasoning and achieve the same efficiency as the reasoning approach for classical method calls handled in Paper 1.

**3: How can we prove the soundness and completeness of our reasoning system?**

The semantics of concurrent asynchronous objects without futures is defined in Paper 1 by using a transformational approach from a syntactical encoding of the different program statements into an underlying sequential language. In each of Papers 2 and 3 an operational semantics is chosen to define the semantics of concurrent asynchronous objects. The reasoning systems in Papers 1 and 2 are defined in Hoare logic, and in Papers 3 and 4 are defined in dynamic logic. We have proved that our reasoning systems are sound and relatively complete with respect to the chosen semantics. Specifications in terms of history invariants may be derived independently for each object and composed in order to derive properties for the global system. The soundness proof of the compositional rule is provided in Paper 3.

**4: How can we realize our reasoning system by means of executable tools?**

Runtime assertion checking and static deductive verification are two approaches chosen in this thesis to check whether concurrent asynchronous objects behave according to their specifications. By using the approach of Papers 2 and 3, we implement the history-explicit semantics of concurrent asynchronous objects in Maude as an extension of the ABS interpreter, by means of a global history reflecting all events that have occurred in the execution. We also extend ABS with method annotations such that ABS programmers can specify invariants, pre- and post-conditions in the code. Future-oriented and history-based properties are expressible in our assertion language and are checked during simulation. KeY is chosen as the formal verification tool in this thesis. We extend KeY with extra rules for proving future-oriented and history-based properties.

In conclusion, to tackle the complexity of distributed systems, we have insisted that the analysis technique in this thesis should be compositional. Accordingly, we develop a sound and relatively complete reasoning system based on a four-event semantics for asynchronous method calls (and first-class futures), which introduces disjoint

alphabets for the local histories of different objects. The processing steps made by one object do not affect the history-based invariants of the surrounding objects. The wellformedness property of global history serves as a connection between the local histories. The global invariant of a total system may be constructed from the history invariants of the composed objects, requiring wellformedness of the global history. Compared to previous work for Creol, this approach allows us to formulate a much simpler proof system for object-oriented language based on concurrent objects communicating by asynchronous method calls (and first-class futures). Furthermore, we provide a runtime assertion checker and an extension of the KeY theorem prover for verifying concurrent asynchronous objects based on our reasoning system.

In these ways we have contributed towards our overall goal which is to simplify reasoning about concurrent objects communicating by asynchronous method calls using history-based invariants and to provide corresponding tool support.

## 6.2 Future Work

History invariants can be naturally included in interface definitions, defining the external visible alphabet of an object and specifying the external behavior of the provided methods. Adding interfaces to our formalism would affect the composition rule in that events not observed through the interface must be hidden.

It will be natural to investigate how our reasoning system would benefit by extending it with rely/guarantee style reasoning. We may for instance (1) use callee interfaces as assumptions in order to express properties of the values or (2) adapt rely/guarantee style reasoning to history invariants [27, 68], for example, the rely part may be expressed as properties over input events, whereas the guaranteed behavior is associated with output events.

Some other object-oriented features such as inheritance is not considered in this work. However, our approach may be combined with behavioral subtyping and lazy behavioral subtyping which has been worked out for the same language setting [46].

At the moment we use explicit quantification in first-order formulae to express ordering constraints over histories for the KeY tool. However, treatment of quantification is a challenge in first-order theorem proving and performance reduces in general significantly with their nesting depth. Therefore, another direction of future work could be to improve the automation of history-based verification in KeY by replacing the use of quantifiers with suitable abstract predicates. Since this approach has been successfully used to support KeY automation for Java strings [19], we believe this approach will also be applicable to *sequences* including communication histories.

# Part II

# Research Papers about Verification of Asynchronously Communicating Objects

APPENDIX A

# Appendix

## A.1 Syntax of the *ABS* functional sublanguage

BNF syntax for the *ABS* functional sublanguage with terms $t$, data type definitions $Dd$, and function definitions $F$ is given below:

| | | | |
|---|---|---|---|
| $Dd$ | ::= | `data` $D$ $\{[Co(T^*)]^*\}$ | data type declaration |
| $F$ | ::= | `def` $T$ $fn([T\ x]^*) == rhs$ | function declaration |
| $t$ | ::= | $Co(e^*) \mid fn([e]^*)$ | constructor and function application |
| | | $\mid (e, e)$ | pair constructor |
| $p$ | ::= | $v \mid Co(p^*) \mid (p, p)$ | pattern |
| $rhs$ | ::= | $e$ | pure expressions |
| | | $\mid$ `case` $e\{b^*\}$ | case expression |
| $b$ | ::= | $p \Rightarrow rhs$ | branch |

Data types are implicitly defined by declaring constructor functions $Co$. The right hand side of the definition of a function $fn$ may be a nested case expression. Patterns include constructor terms and pairs over constructor terms. The functional if-then-else construct and infix operators are not included in the syntax above. We use + and − for numbers, **and** and **or** for booleans, and = for equality.

## A.2 Complete Code of Fairness Reader/Writer

```
data Data{int(Int) bool(Bool) string(String) obj(Obj) Nothing}
data Map{Empty Bind(Int, Data, Map)}
data DataSet{Empty Add(Data, DataSet)}

def Bool isElement(Data element, DataSet set) ==
  case set{Empty => False;
    Add(d, s) => element = d or isElement(element, s)}

def Data lookup(Int key, Map map) ==
  case map{Empty => Nothing;
    Bind(k, d, m) => if key = k then d else lookup(key, m) }

def DataSet delete(Data element, DataSet set) ==
```

```
    case set{Empty => Empty;
       Add(d, s) => if element = d then delete(element, s) else Add(d, delete(element, s))}

def Map modify(Int key, Data element, Map map) ==
    case map{Empty => Bind(key, element, Empty);
       Bind(k, d, m) => if key = k then Bind(k, element, m)
                        else Bind(k, d, modify(key, element, m))}

def Int size(DataSet set) ==
    case set{Empty => 0;
       Add(d, s) => 1 + size(s)}

interface RW{
    Void openR();
    Void closeR();
    Void openW();
    Void closeW();
    Data read(Int key);
    Void write(Int key, Data element) }

interface DB{
    Data read(Int key);
    Void write(Int key, Data element)}

class DataBase implements DB{
    Map map;
    {map := Empty;}
    Data read(Int key) {return lookup(key, map)}
    Void write(Int key, Data element) {map := modify(key, element, map)} }

class RWController() implements RW{
    DB db; DataSet readers; Obj writer; Int pr;
    {db := new DataBase(); readers := Empty; writer := null; pr := 0}
    Void openR(){await writer = null; readers := Add(caller, readers)}
    Void closeR(){readers := delete(caller, readers)}
    Void openW(){await writer = null; writer := caller; readers := Add(caller, readers)}
    Void closeW(){await writer = caller; writer := null; readers := delete(caller, readers)}
    Data read(Int key){ Data result;
       await isElement(caller, readers); pr := pr + 1;
       await result := db.read(key); pr := pr − 1; return result }
    Void write(Int key, Data value){
       await caller = writer and pr = 0 and
         (readers = Empty or (isElement(writer, readers) and size(readers) = 1));
       db.write(key, value) }}
```

## A.3 Definition of Writers

$Writers : Seq[Ev] \to Set[Obj]$

$Writers(\varepsilon) \triangleq \emptyset$
$Writers(h \vdash o \leftarrow \mathsf{this.openW}) \triangleq Writers(h) \cup \{o\}$
$Writers(h \vdash o \leftarrow \mathsf{this.closeW}) \triangleq Writers(h) \setminus \{o\}$
$Writers(h \vdash \boldsymbol{others}) \triangleq Writers(h)$

## A.4 Definition of Writing

$Writing : Seq[Ev] \to Nat$

$Writing(h) \triangleq \#(h/\{\mathsf{this} \rightarrow \mathsf{db.write}\}) - \#(h/\{\mathsf{this} \twoheadleftarrow \mathsf{db.write}\})$

## A.5 Verification Details for **RWController**

### A.5.1 Method: openR

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$ :

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
`await` $\mathsf{writer} = \mathsf{null}$;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge$
$\mathsf{writer} = \mathsf{null}\}$
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{Add}(\mathsf{caller}, \mathsf{readers}) \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$
$\mathsf{readers} := \mathsf{Add}(\mathsf{caller}, \mathsf{readers})$;
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$

### A.5.2 Method: openW

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$ :

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
`await` $\mathsf{writer} = \mathsf{null}$;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge$
$\mathsf{writer} = \mathsf{null}\}$
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{Add}(\mathsf{caller}, \mathsf{readers}) \wedge$
$Writers(\mathcal{H}) \cup \{\mathsf{caller}\} = \{\mathsf{caller}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$
$\mathsf{writer} := \mathsf{caller}$;
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{Add}(\mathsf{caller}, \mathsf{readers}) \wedge$
$Writers(\mathcal{H}) \cup \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$
$\mathsf{readers} := \mathsf{Add}(\mathsf{caller}, \mathsf{readers})$;
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{readers} \wedge$
$Writers(\mathcal{H}) \cup \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$

### A.5.3 Method: closeR

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$ :

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = delete(\mathsf{caller}, \mathsf{readers}) \wedge$
$Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
$\mathsf{readers} := delete(\mathsf{caller}, \mathsf{readers})$;
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$

### A.5.4 Method: closeW

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$ :

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
`await` writer $=$ caller;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})$
$\quad \wedge \mathsf{writer} = \mathsf{caller}\}$
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = delete(\mathsf{caller}, \mathsf{readers}) \wedge$
$\quad Writers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \{\mathsf{null}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
writer $:= null$;
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = delete(\mathsf{caller}, \mathsf{readers}) \wedge$
$\quad Writers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
readers $:= delete(\mathsf{caller}, \mathsf{readers})$;
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$

### A.5.5 Method: read

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$ :

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
`await` isElement(caller, readers);
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \mathsf{isElement}(\mathsf{caller}, \mathsf{readers})\}$
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) + 1 = \mathsf{pr} + 1 \wedge OK(\mathcal{H}) \wedge Writing(\mathcal{H}) = 0\}$
pr $:=$ pr $+ 1$;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) + 1 = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge Writing(\mathcal{H}) = 0\}$
`await` result $:=$ db.read(key);
$\{(\exists \mathsf{result} \ . \ (I_1 \wedge I_2 \wedge I_3 \wedge I_4)^{\mathcal{H}}_{pop(\mathcal{H})}) \wedge \mathcal{H}$ `ew` this $\twoheadleftarrow db.\mathsf{read}(\mathsf{result})\}$
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} - 1 \wedge OK(\mathcal{H})\}$
pr $:=$ pr $- 1$;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
`return` result;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$

### A.5.6 Method: write

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$ :

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
`await` caller $=$ writer && pr $= 0$ &&

$(\mathsf{readers} = \mathit{Empty} \vee (\mathsf{isElement}(\mathsf{writer}, \mathsf{readers}) \&\& \mathsf{size}(\mathsf{readers}) = 1))$;
$\{\mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H}) \wedge$
$\quad \mathsf{caller} = \mathsf{writer} \wedge \mathsf{pr} = 0 \wedge (\mathsf{readers} = \mathit{Empty} \vee$
$\quad (\mathsf{isElement}(\mathsf{writer}, \mathsf{readers}) \wedge \mathsf{size}(\mathsf{readers}) = 1))\}$
$\{\mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge$
$\quad \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H}) \wedge \mathit{Reading}(\mathcal{H}) = 0 \wedge$
$\quad \#\mathit{Writers}(\mathcal{H}) = 1\}$
$\mathsf{db.write}(\mathsf{key}, \mathsf{value})$;
$\{\mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H})\}$

## A.6 Verification Details for Unbounded Buffer

### A.6.1 The put method

Proof outline:

$\{I\}$
$\{((\mathsf{cnt} = 0 \Rightarrow Q_x^{\mathsf{cell}}) \wedge (\mathsf{cnt} \neq 0 \Rightarrow Q^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\rightarrow \mathsf{next}.\mathit{put}(x) \vdash \mathsf{this}\leftarrow \mathsf{next}.\mathit{put}}))^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\twoheadrightarrow \mathsf{this}.\mathit{put}(x)}\}$
$\mathcal{H} = \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.\mathit{put}(\overline{x})$;
$\{(\mathsf{cnt} = 0 \Rightarrow Q_x^{\mathsf{cell}}) \wedge (\mathsf{cnt} \neq 0 \Rightarrow Q^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\rightarrow \mathsf{next}.\mathit{put}(x) \vdash \mathsf{this}\leftarrow \mathsf{next}.\mathit{put}})\}$
`if` $(\mathsf{cnt} = 0)$ `then` $\{Q_x^{\mathsf{cell}}\}$ $\mathsf{cell} := x$
`else if` $(\mathsf{next} = \mathsf{null})$ `then` $\mathsf{next} :=$ `new` Buffer `fi`;
$\{\mathsf{next} \neq \mathsf{null} \wedge Q^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\rightarrow \mathsf{next}.\mathit{put}(x) \vdash \mathsf{this}\leftarrow \mathsf{next}.\mathit{put}}\}$
$\mathsf{next.put}(x)$
`fi`;
$\{Q\}$
$\mathsf{cnt} := \mathsf{cnt} + 1$;
$\{I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\leftarrow \mathsf{this}.\mathit{put}}\}$
$\mathcal{H} = \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.\mathit{put}$;
$\{I\}$

where $Q \triangleq I^{\mathcal{H},\mathsf{cnt}}_{\mathcal{H}\vdash \mathsf{caller}\leftarrow \mathsf{this}.\mathit{put},\mathsf{cnt}+1}$

The proof outline leads to two verification conditions:

$$(1) \quad I \wedge \mathsf{cnt} = 0 \Rightarrow (Q_x^{\mathsf{cell}})^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\twoheadrightarrow \mathsf{this}.\mathit{put}(x)}$$
$$(2) \quad I \wedge \mathsf{cnt} \neq 0 \Rightarrow (Q^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\rightarrow \mathsf{next}.\mathit{put}(x) \vdash \mathsf{this}\leftarrow \mathsf{next}.\mathit{put}})^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\twoheadrightarrow \mathsf{this}.\mathit{put}(x)}$$

**Invariant Analysis**

The two class invariants are proved by the following verification conditions:

$I_1 : \mathsf{cnt} = \#(\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(1) :

$\mathsf{cnt} = \#(\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} = 0$
$\Rightarrow$
$\mathsf{cnt} + 1 = \#(x \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$(2):$
$\mathsf{cnt} = \#(\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} \neq 0$
$\Rightarrow$
$\mathsf{cnt} + 1 = \#(\mathsf{cell} \dashv (\mathit{in}(\mathsf{next}, h) \vdash x \texttt{ after } \#\mathit{out}(\mathsf{next}, h)))$

$I_2 : \mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$(1):$
$I_1 \wedge (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))) \wedge \mathsf{cnt} = 0$
$\Rightarrow$
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) \vdash x = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (x \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$(2):$
$I_1 \wedge (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))) \wedge \mathsf{cnt} \neq 0$
$\Rightarrow$
$(\mathit{out}(\mathsf{next}, \mathcal{H}) \leq \mathit{in}(\mathsf{next}, \mathcal{H}) \vdash x) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) \vdash x = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}) \vdash x)$

### A.6.2 The get method

Proof outline:

$\{I\}$
$\{I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.\mathit{get}}\}$
$\mathcal{H} = \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.\mathit{get};$
$\{I\}$ `var` Obj $r$; $\{I\}$ `await`$(\mathsf{cnt} > 0)$; $\{I \wedge \mathsf{cnt} > 0\}$
$\{((\mathsf{cell} = \mathsf{null} \Rightarrow \forall r'.Q^{r,\mathcal{H}}_{r',\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{next}.\mathit{get} \vdash \mathsf{this} \twoheadleftarrow \mathsf{next}.\mathit{get}(r')}) \wedge (\mathsf{cell} \neq \mathsf{null} \Rightarrow (Q^{\mathsf{cell}}_{\mathsf{null}})^{r}_{\mathsf{cell}}))^{\mathsf{cnt}}_{\mathsf{cnt}-1}\}$
$\mathsf{cnt} := \mathsf{cnt} - 1;$
$\{(\mathsf{cell} = \mathsf{null} \Rightarrow \forall r'.Q^{r,\mathcal{H}}_{r',\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{next}.\mathit{get} \vdash \mathsf{this} \twoheadleftarrow \mathsf{next}.\mathit{get}(r')}) \wedge (\mathsf{cell} \neq \mathsf{null} \Rightarrow (Q^{\mathsf{cell}}_{\mathsf{null}})^{r}_{\mathsf{cell}})\}$
`if`$(\mathsf{cell} = \mathsf{null})$ `then` $\{\forall r'.Q^{r,\mathcal{H}}_{r',\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{next}.\mathit{get} \vdash \mathsf{this} \twoheadleftarrow \mathsf{next}.\mathit{get}(r')}\}$ $r := \mathsf{next}.\mathit{get}()$
`else` $\{(Q^{\mathsf{cell}}_{\mathsf{null}})^{r}_{\mathsf{cell}}\}$ $r := \mathsf{cell}; \mathsf{cell} := \mathsf{null}$ `fi`;
$\{Q\}$
`return` $r$;
$\{I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.\mathit{get}(r)}\}$
$\mathcal{H} = \mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.\mathit{get}(r);$
$\{I\}$

where $Q \triangleq I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.\mathit{get}(r)}$

The proof outline leads to three verification conditions:

$$(1)\quad I \Rightarrow I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\rightarrow\mathsf{this}.get}$$
$$(2)\quad I \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} = \mathsf{null} \Rightarrow (\forall r'.Q^{r,\mathcal{H}}_{r',\mathcal{H}\vdash \mathsf{this}\rightarrow\mathsf{next}.get\vdash \mathsf{this}\leftarrow\mathsf{next}.get(r')})^{\mathsf{cnt}}_{\mathsf{cnt}-1}$$
$$(3)\quad I \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} \neq \mathsf{null} \Rightarrow ((Q^{\mathsf{cell}}_{\mathsf{null}})^{r}_{\mathsf{cell}})^{\mathsf{cnt}}_{\mathsf{cnt}-1}$$

**Invariant Analysis**

The two class invariants are proved by the following verification conditions:

$I_1 : \mathsf{cnt} = \#(\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$(1)$ :
$\mathsf{cnt} = \#(\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$\mathsf{cnt} = \#(\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$(2)$ :
$\mathsf{cnt} = \#(\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} = \mathsf{null}$
$\Rightarrow$
$\mathsf{cnt} - 1 = \#(\mathsf{cell} \dashv (\mathit{in}(\mathsf{next}, h)\ \texttt{after}\ \#(\mathit{out}(\mathsf{next}, h) \vdash x)))$
$(3)$ :
$\mathsf{cnt} = \#(\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} \neq \mathsf{null}$
$\Rightarrow$
$\mathsf{cnt} - 1 = \#(\mathsf{null} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$

$I_2 : \mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$(1)$ :
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$(2)$ :
$I_1 \wedge (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))) \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} = \mathsf{null}$
$\Rightarrow$
$(\mathit{out}(\mathsf{next}, \mathcal{H}) \vdash r \leq \mathit{in}(\mathsf{next}, \mathcal{H})) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = (\mathit{out}(\mathsf{this}, \mathcal{H}) \vdash r) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{rest}(\mathit{buf}(\mathsf{next}, \mathcal{H})))$
$(3)$ :
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} \neq \mathsf{null}$
$\Rightarrow$
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = (\mathit{out}(\mathsf{this}, \mathcal{H}) \vdash \mathsf{cell}) \ \vdash\!\dashv (\mathsf{null} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$

### A.6.3 The conditional FIFO property

We derive the conditional FIFO property, $\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{fifo}(\mathsf{this}, \mathcal{H})$, from the assumption of class invariant (7.7)

$$\mathit{fifo}(\mathsf{next},\mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this},\mathcal{H}) = \mathit{out}(\mathsf{this},\mathcal{H}) \ \vdash (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next},\mathcal{H}))$$
$$\Rightarrow$$
$$\mathit{fifo}(\mathsf{next},\mathcal{H}) \Rightarrow \ \mathit{fifo}(\mathsf{this},\mathcal{H})$$

$$\mathit{in}(\mathsf{this},\mathcal{H}) = \mathit{out}(\mathsf{this},\mathcal{H}) \ \vdash (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next},\mathcal{H}))$$
$$\Rightarrow$$
$$\mathit{out}(\mathsf{this},\mathcal{H}) \leq \mathit{in}(\mathsf{this},\mathcal{H})$$

### A.6.4 Verification of the History Invariant

Here we consider the details for verifying the conditional FIFO property, named $\mathit{Cond}_{\mathit{fifo}}$, as a history invariant. The invariant is formulated as: $\mathit{fifo}(\mathsf{next},\mathcal{H}) \Rightarrow \mathit{fifo}(\mathsf{this},\mathcal{H})$ A history invariant, $I_{\mathsf{this}:C(\overline{cp})}$, can be verified by showing that it is maintained by each local statement $s$ affecting the history, i.e., one must prove $\{I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H}) \wedge P\}\ s\ \{Q \Rightarrow I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H})\}$ where $P$ and $Q$ are the pre- and postconditions of $s$ used in the proof outline of the method.

#### The put method

In the above sections, we have proved that $\mathit{Cond}_{\mathit{fifo}}$ follows by implication from the class invariant, and that the class invariant holds at method termination. Thus, it remains to prove that $\mathit{Cond}_{\mathit{fifo}}$ holds after each history extension inside the method body, i.e., we must prove that the property holds after next.put(x):

$$\{\mathit{Cond}_{\mathit{fifo}} \wedge P\}\ \mathsf{next.put(x)}\ \{Q \Rightarrow \mathit{Cond}_{\mathit{fifo}}\}$$

where
$Q \triangleq (\mathit{fifo}(\mathsf{next},\mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this},\mathcal{H}) = \mathit{out}(\mathsf{this},\mathcal{H}) \ \vdash (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next},\mathcal{H})))^{\mathcal{H},\mathsf{cnt}}_{\mathcal{H}\vdash \mathsf{caller}\leftarrow \mathsf{this}.\mathit{put},\mathsf{cnt}+1}$
the actual assertion $P$ is not needed here, but it is given in the proof outline above. The postcondition: $Q \Rightarrow \mathit{Cond}_{\mathit{fifo}}$ is proved as follows:

$$(\mathit{fifo}(\mathsf{next},\mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this},\mathcal{H}) = \mathit{out}(\mathsf{this},\mathcal{H}) \ \vdash (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next},\mathcal{H})))$$
$$\Rightarrow$$
$$(\mathit{fifo}(\mathsf{next},\mathcal{H}) \Rightarrow \ \mathit{fifo}(\mathsf{this},\mathcal{H}))$$

$$\mathit{in}(\mathsf{this},\mathcal{H}) = \mathit{out}(\mathsf{this},\mathcal{H}) \ \vdash (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next},\mathcal{H}))$$
$$\Rightarrow$$
$$\mathit{out}(\mathsf{this},\mathcal{H}) \leq \mathit{in}(\mathsf{this},\mathcal{H})$$

#### The get method

In the above sections, we have proved that $\mathit{Cond}_{\mathit{fifo}}$ follows by implication from the class invariant, and that the class invariant holds at method termination. Thus, it

remains to prove that $Cond_{fifo}$ holds after each history extension inside the method body, i.e., we must prove that the property holds after r:=next.get():

$$\{Cond_{fifo} \wedge P\}\ \mathsf{r{:=}next.get()}\ \{Q \Rightarrow Cond_{fifo}\}$$

where
$Q \triangleq (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H})\ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H})))^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller} \leftarrow \mathsf{this}.get(r)}$
the actual assertion $P$ is not needed here, but it is given in the proof outline above.
$(\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \vdash r\ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H})))$
$\Rightarrow$
$(\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow\ \mathit{fifo}(\mathsf{this}, \mathcal{H}))$

$\mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \vdash r\ \vdash\!\dashv (\mathsf{cell} \dashv \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$\mathit{out}(\mathsf{this}, \mathcal{H}) \leq \mathit{in}(\mathsf{this}, \mathcal{H})$

## A.7 Proof of Lemma 7.6

For an assertion $P$, we let $P'$ abbreviate $P^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}$, and $I$ abbreviates $I_C$.

### A.7.1 Soundness

For the rules (NOTNULL), (RETURN), (SUSPEND), (AWAIT), (CALLASYNC), (CALLSYNC1), (CALLSYNC1-1), (AWAITCALL1), (AWAITCALL2), and (NEW)which are of the form $\{P\}\ s\ \{Q\}$, soundness follow directly from the *wlp*, i.e., for each rule the formula $P \Rightarrow \mathit{wlp}(s, Q)$ holds.

**Rule (method)** Let $s' \triangleq \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}); s; \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return})$

The rule may then be formulated as:

$$(\text{METHOD})\quad \frac{\{S\}\ s'\ \{\mathit{wf}(\mathcal{H}) \Rightarrow R\}}{\{\forall \overline{y}\,.\,S\}\ m(\overline{x})\{\mathtt{var}\ \overline{y}; s\}\ \{\exists \overline{y}\,.\,R\}}$$

and we have the following *wlp*:

$$\mathit{wlp}(m(\overline{x})\{\mathtt{var}\ \overline{y}; s\}, Q) \triangleq\ \mathit{wlp}(\mathtt{var}\ \overline{y}; s', \mathit{wf}(\mathcal{H}) \Rightarrow Q)$$

where $\overline{y} \notin FV[Q]$.

For soundness, we need to ensure

$$\forall \overline{y}\,.\,S \Rightarrow \mathit{wlp}(\mathtt{var}\ \overline{y}; s', \mathit{wf}(\mathcal{H}) \Rightarrow \exists \overline{y}\,.\,R)$$

under the assumption $S \Rightarrow \mathit{wlp}(s', \mathit{wf}(\mathcal{H}) \Rightarrow R)$ (rule premise). Since $R \Rightarrow \exists \overline{y}\,.\,R$, we have $\mathit{wlp}(s', \mathit{wf}(\mathcal{H}) \Rightarrow R) \Rightarrow \mathit{wlp}(s', \mathit{wf}(\mathcal{H}) \Rightarrow \exists \overline{y}\,.\,R)$. Thus it suffices to prove $\forall \overline{y}\,.\,S \Rightarrow \mathit{wlp}(\mathtt{var}\ \overline{y}, S)$ which is trivial.

**Rule (callSync2)** Let $i$ denote the event $\mathsf{this} \to o.m(\overline{e})$, and $c_{(o,v)}$ denote the event $\mathsf{this} \twoheadleftarrow o.m(v)$. We have the following proof obligation:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this} \Rightarrow wlp(v := o.m(\overline{e}), \exists z \,.\, R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v)})$$

under the assumption $S \wedge \mathsf{caller} = \mathsf{this} \Rightarrow wlp(body, R \wedge \mathsf{caller} = \mathsf{this})$ (rule premise). Here, the $wlp$ is defined by:

$$wlp(v := o.m(\overline{e}), Q) \triangleq o = \mathsf{this} \Rightarrow (wlp(body, Q^{v,\mathcal{H},\overline{y},v'}_{v',\mathcal{H}\vdash c_{(\mathsf{this},v')},\overline{y}',\mathsf{return}}))^{\overline{x},\mathsf{caller},\overline{y}',\mathcal{H}}_{\overline{e},\mathsf{this},\overline{y},\mathcal{H}\vdash i}$$

which means that the above proof obligation can be written as:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \Rightarrow (wlp(body, (\exists z \,.\, R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v)})^{v,\mathcal{H},\overline{y},v'}_{v',\mathcal{H}\vdash c_{(\mathsf{this},v')},\overline{y}',\mathsf{return}}))^{\overline{x},\mathsf{caller},\overline{y}',\mathcal{H}}_{\overline{e},\mathsf{this},\overline{y},\mathcal{H}\vdash i}$$

Remark that $S$ and $R$ are assertions over the state of the called method, i.e., $\overline{y}$ and $\overline{y}'$ does not occur in these assertions. Since $\overline{y} \notin FV[R]$, we have the following implication:

$$R \wedge \mathsf{caller} = \mathsf{this} \Rightarrow (\exists z \,.\, R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v)})^{v,\mathcal{H},\overline{y},v'}_{v',\mathcal{H}\vdash c_{(\mathsf{this},v')},\overline{y}',\mathsf{return}}$$

Since $wlp$ is monotonic, it therefore suffices to prove:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \Rightarrow (wlp(body, R \wedge \mathsf{caller} = \mathsf{this}))^{\overline{x},\mathsf{caller},\overline{y}',\mathcal{H}}_{\overline{e},\mathsf{this},\overline{y},\mathcal{H}\vdash i}$$

which follows by rule premise and the trivial implication:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \Rightarrow (S \wedge \mathsf{caller} = \mathsf{this})^{\overline{x},\mathsf{caller},\overline{y}',\mathcal{H}}_{\overline{e},\mathsf{this},\overline{y},\mathcal{H}\vdash i}$$

**Rule (callSync2-1)** This proof follows the same pattern as for (callSync2). As above we have $\overline{y}$ and $\overline{y}'$ not in $FV[S]$ and $FV[R]$, and we therefore ignore these variables below. Here we have the proof obligation:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this} \Rightarrow wlp(o.m(\overline{e}), \exists v' \,.\, R^{\mathsf{caller},\mathsf{return},\mathcal{H}}_{\mathsf{this},v',pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v')})$$

under the assumption $S \wedge \mathsf{caller} = \mathsf{this} \Rightarrow wlp(body, R \wedge \mathsf{caller} = \mathsf{this})$ (rule premise). Here, the $wlp$ is defined by:

$$wlp(o.m(\overline{e}), Q) \triangleq o = \mathsf{this} \Rightarrow (wlp(body, Q^{\mathcal{H},v'}_{\mathcal{H}\vdash c_{(\mathsf{this},v')},\mathsf{return}}))^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i}$$

Since

$$R \wedge \mathsf{caller} = \mathsf{this} \Rightarrow (\exists v' \,.\, R^{\mathsf{caller},\mathsf{return},\mathcal{H}}_{\mathsf{this},v',pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v')})^{\mathcal{H},v'}_{\mathcal{H}\vdash c_{(\mathsf{this},v')},\mathsf{return}}$$

The proof obligation reduces to

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \Rightarrow (wlp(body, R \wedge \mathsf{caller} = \mathsf{this}))^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i}$$

which is satisfied by the same argument as above.

### A.7.2 Completeness

Statement `suspend`.

1. $\{I\}$ `suspend` $\{I\}$ (SUSPEND)
2. $\{\mathcal{H} = h_0\}$ `suspend` $\{h_0 \leq \mathcal{H}\}$ (HIS)
3. $\{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$ (WF)
4. $\{I \wedge \mathcal{H} = h_0 \wedge wf(\mathcal{H})\}$ `suspend` $\{I \wedge h_0 \leq \mathcal{H} \wedge wf(\mathcal{H})\}$ $(1, 2, 3, (\text{CONJ}))$
5. $\{\forall \overline{w}', \mathcal{H}' . (\forall h_0 . I \wedge \mathcal{H} = h_0 \wedge wf(\mathcal{H}) \Rightarrow I' \wedge h_0 \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ `suspend` $\{Q\}$ $(4, (\text{ADAP}))$
6. $\{\forall \overline{w}', \mathcal{H}' . (I \wedge wf(\mathcal{H}) \Rightarrow I' \wedge \mathcal{H} \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ `suspend` $\{Q\}$ $(5, math)$
7. $\{I \wedge wf(\mathcal{H}) \wedge \forall \overline{w}', \mathcal{H}' . (I' \wedge \mathcal{H} \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ `suspend` $\{Q\}$ $(6, (\text{CONS}))$
8. $\{wlp(\texttt{suspend}, Q)\}$ `suspend` $\{Q\}$ $(7, def)$

Statement `await` $b$.

1. $\{I\}$ `await` $b\ \{I \wedge b\}$ (AWAIT)
2. $\{\mathcal{H} = h_0\}$ `await` $b\ \{h_0 \leq \mathcal{H}\}$ (HIS)
3. $\{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$ (WF)
4. $\{I \wedge \mathcal{H} = h_0 \wedge wf(\mathcal{H})\}$ `await` $b\ \{I \wedge b \wedge h_0 \leq \mathcal{H} \wedge wf(\mathcal{H})\}$ $(1, 2, 3, (\text{CONJ}))$
5. $\{\forall \overline{w}', \mathcal{H}' . (\forall h_0 . I \wedge \mathcal{H} = h_0 \wedge wf(\mathcal{H}) \Rightarrow I' \wedge b' \wedge h_0 \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ `await` $b\ \{Q\}$ $(4, (\text{ADAP}))$
6. $\{\forall \overline{w}', \mathcal{H}' . (I \wedge wf(\mathcal{H}) \Rightarrow I' \wedge b' \wedge \mathcal{H} \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ `await` $b\ \{Q\}$ $(5, math)$
7. $\{I \wedge wf(\mathcal{H}) \wedge \forall \overline{w}', \mathcal{H}' . (I' \wedge b' \wedge \mathcal{H} \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ `await` $b\ \{Q\}$ $(6, (\text{CONS}))$
8. $\{wlp(\texttt{await}\ b, Q)\}$ `await` $b\ \{Q\}$ $(7, def)$

Statement $\texttt{await}\ o.m(\overline{e})$.

1. $\{h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0\}$
   $\texttt{await}\ o.m(\overline{e})$
   $\{h_0 \leq \mathcal{H} \wedge I^{\mathcal{H}}_{pop(\mathcal{H})} \wedge \exists v\,.\,\mathcal{H}\ \texttt{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v)\}$ (awaitCall2)
2. $\{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$ (wf)
3. $\{h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0 \wedge wf(\mathcal{H})\}$
   $\texttt{await}\ o.m(\overline{e})$
   $\{h_0 \leq \mathcal{H} \wedge I^{\mathcal{H}}_{pop(\mathcal{H})} \wedge \exists v\,.\,\mathcal{H}\ \texttt{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v) \wedge wf(\mathcal{H})\}$ (1, 2, (conj))
4. $\{\forall \overline{w}', \mathcal{H}'\,.\,(\forall h_0, o_0\,.\,(h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0 \wedge wf(\mathcal{H}))$
   $\Rightarrow h_0 \leq \mathcal{H}' \wedge (I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge \exists v\,.\,\mathcal{H}'\ \texttt{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v) \wedge wf(\mathcal{H}'))$
   $\Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}\}\texttt{await}\ o.m(\overline{e})\{Q\}$ (3, (adap))
5. $\{\forall \overline{w}', \mathcal{H}'\,.\,(I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\to o.m(\overline{e})} \wedge wf(\mathcal{H}) \Rightarrow$
   $\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge (I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge \exists v\,.\,\mathcal{H}'\ \texttt{ew}\ \mathsf{this} \twoheadleftarrow o.m(v)$
   $\wedge wf(\mathcal{H}')) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}\}\texttt{await}\ o.m(\overline{e})\{Q\}$ (4, *math*)
6. $\{I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $(I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge \exists v\,.\,\mathcal{H}'\ \texttt{ew}\ \mathsf{this} \twoheadleftarrow o.m(v) \wedge wf(\mathcal{H}')) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}\}$
   $\texttt{await}\ o.m(\overline{e})\{Q\}$ (5, (cons))
7. $\{I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $I^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'\vdash \mathsf{this}\twoheadleftarrow o.m(v')}\}$
   $\texttt{await}\ o.m(\overline{e})\{Q\}$ (6, (cons))
8. $\{o = \mathsf{null}\}\ \texttt{await}\ o.m(\overline{e})\ \{\mathit{false}\}$ (notNull)
9. $\{o \neq \mathsf{null} \Rightarrow (I_C{}^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $I^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'\vdash \mathsf{this}\twoheadleftarrow o.m(v')})\}$
   $\texttt{await}\ o.m(\overline{e})\{Q\}$ (7, 8, (disj))
10. $\{wlp(\texttt{await}\ o.m(\overline{e}), Q)\}\ \texttt{await}\ o.m(\overline{e})\ \{Q\}$ (9, *def*)

Statement `await` $v := o.m(\overline{e})$.

1. $\{h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0\}$
   `await` $v := o.m(\overline{e})$
   $\{h_0 \leq \mathcal{H} \wedge \mathcal{H}\ \mathtt{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v) \wedge \exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H})}\}$ (awaitCall1)
2. $\{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$ (wf)
3. $\{h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0 \wedge wf(\mathcal{H})\}$
   `await` $v := o.m(\overline{e})$
   $\{h_0 \leq \mathcal{H} \wedge \mathcal{H}\ \mathtt{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v) \wedge \exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H})} \wedge wf(\mathcal{H})\}$ $(1, 2, (\text{conj}))$
4. $\{\forall v', \overline{w}', \mathcal{H}'\,.\,(\forall h_0, o_0\,.\,(h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0 \wedge wf(\mathcal{H}))$
   $\Rightarrow h_0 \leq \mathcal{H}' \wedge \mathcal{H}'\ \mathtt{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v') \wedge (\exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge wf(\mathcal{H}'))$
   $\Rightarrow Q^{v,\overline{w},\mathcal{H}}_{v',\overline{w}',\mathcal{H}'}\}$`await` $v := o.m(\overline{e})\{Q\}$ $(3, (\text{adap}))$
5. $\{\forall v', \overline{w}', \mathcal{H}'\,.\,(I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\to o.m(\overline{e})} \wedge wf(\mathcal{H})$
   $\Rightarrow \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge \mathcal{H}'\ \mathtt{ew}\ \mathsf{this} \twoheadleftarrow o.m(v') \wedge$
   $(\exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge wf(\mathcal{H}')) \Rightarrow Q^{v,\overline{w},\mathcal{H}}_{v',\overline{w}',\mathcal{H}'}\}$`await` $v := o.m(\overline{e})\{Q\}$ $(4, math)$
6. $\{I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge \mathcal{H}'\ \mathtt{ew}\ \mathsf{this} \twoheadleftarrow o.m(v') \wedge$
   $(\exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge wf(\mathcal{H}')) \Rightarrow Q^{v,\overline{w},\mathcal{H}}_{v',\overline{w}',\mathcal{H}'}\}$`await` $v := o.m(\overline{e})\{Q\}$ $(5, (\text{cons}))$
7. $\{I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $I^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \Rightarrow (Q^{v}_{v'})^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'\vdash \mathsf{this}\twoheadleftarrow o.m(v')}\}$
   `await` $v := o.m(\overline{e})\{Q\}$ $(6, (\text{cons}))$
8. $\{o = \mathsf{null}\}$ `await` $v := o.m(\overline{e})\ \{false\}$ (notNull)
9. $\{o \neq \mathsf{null} \Rightarrow (I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $I^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \Rightarrow (Q^{v}_{v'})^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'\vdash \mathsf{this}\twoheadleftarrow o.m(v')})\}$
   `await` $v := o.m(\overline{e})\{Q\}$ $(7, 8, (\text{disj}))$
10. $\{wlp(\mathtt{await}\ v := o.m(\overline{e}), Q)\}$ `await` $v := o.m(\overline{e})\ \{Q\}$ $(9, def)$

Statement $v := o.m(\overline{e})$.
Let $i$ abbreviate the event $\mathsf{this} \to o.m(\overline{e})$ (which equals $\mathsf{this} \to \mathsf{this}.m(\overline{e})$ under the assumption $\mathsf{this} = o$), and let $c_{(o,v)}$ abbreviate the event $\mathsf{this} \twoheadleftarrow o.m(v)$. Given an arbitrary postcondition $Q$ to the statement $v := o.m(\overline{e})$, i.e., $FV[Q] \subseteq \{\overline{w}, \mathcal{H}, \overline{cp}, \overline{y}, \overline{l}\}$ where $\overline{y}$ are the method-local variables of the caller (including the formal parameters and $\mathsf{caller}$) and $\overline{l}$ is a list of logical variables. Observe that $\mathsf{return} \notin FV[Q]$ since $Q$ appears inside the body of the calling method. By definition, the assertion $wlp(v := o.m(\overline{e}), Q)$ may then be written as:

$$o \neq \mathsf{null} \Rightarrow \forall v'\,.\,\mathtt{if}\ o = \mathsf{this}\ \mathtt{then}\ (wlp(v' := m'(\overline{e}, \mathsf{this}), Q^{v,\mathcal{H}}_{v',\mathcal{H}\vdash c_{(\mathsf{this},v')}}))^{\mathcal{H}}_{\mathcal{H}\vdash i}$$
$$\mathtt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\vdash i\vdash c_{(o,v')}}$$

which by definition of $wlp(v' := m'(\overline{e}, \mathsf{this}), Q)$ can be rewritten as:

$$o \neq \mathsf{null} \Rightarrow \forall v'\,.\,\mathtt{if}\ o = \mathsf{this}\ \mathtt{then}$$
$$((wlp(m'(\overline{x}, \mathsf{caller})\ body, (Q^{v,\mathcal{H}}_{v',\mathcal{H}\vdash c_{(\mathsf{this},v')}})^{\overline{y},v'}_{\overline{y}',\mathsf{return}}))^{\overline{x},\mathsf{caller},\overline{y}'}_{\overline{e},\mathsf{this},\overline{y}})^{\mathcal{H}}_{\mathcal{H}\vdash i}$$
$$\mathtt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\vdash i\vdash c_{(o,v')}}$$

By simplifying the substitutions, this formula may be written as:

$$o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \texttt{if}\ o = \mathsf{this}\ \texttt{then}\ (wlp(m(\overline{x})body, Q^{v,\mathcal{H},\overline{y}}_{\mathsf{return},\mathcal{H}\vdash c_{(\mathsf{this},\mathsf{return})},\overline{y}'}))^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}}\ \texttt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\vdash i\vdash c_{(o,v')}}$$

In the proof below, we let $P$ denote the following assertion:

$$(wlp(m(\overline{x})\ body, Q^{v,\mathcal{H},\overline{y}}_{\mathsf{return},\mathcal{H}\vdash c_{(\mathsf{this},\mathsf{return})},\overline{y}'}))^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}}$$

which means that the *wlp* can be written as:

$$wlp(v := o.m(\overline{e}), Q)\ \triangleq\ o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \texttt{if}\ o = \mathsf{this}\ \texttt{then}\ P\ \texttt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\vdash i\vdash c_{(o,v')}}$$

In the proof below, we let $S$ denote the formula $wlp(m(\overline{x})\ body, Q^{v,\mathcal{H},\overline{y}}_{\mathsf{return},\mathcal{H}\vdash c_{(\mathsf{this},\mathsf{return})},\overline{y}'})$, and $R$ denote the formula $Q^{v,\mathcal{H},\overline{y}}_{\mathsf{return},\mathcal{H}\vdash c_{(\mathsf{this},\mathsf{return})}\overline{y}'}$.

1. $\{S\}\ m(\overline{x})body\ \{R\}$ (*premise*)
2. $\{S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this}\}$
   $v := o.m(\overline{e})\{\exists z \,.\, R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \texttt{ew}\ c_{(\mathsf{this},v)}\}$ (1, (CALLSYNC2))
3. $\{\forall \overline{w}', \mathcal{H}', v' \,.\, (\forall \overline{y}', \bar{l} \,.\, S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this} \Rightarrow$
   $(\exists z \,.\, R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \texttt{ew}\ c_{(\mathsf{this},v)})^{\overline{w},\mathcal{H},v}_{\overline{w}',\mathcal{H}',v'}) \Rightarrow Q^{\overline{w},\mathcal{H},v}_{\overline{w}',\mathcal{H}',v'}\}$
   $v := o.m(\overline{e})\{Q\}$ (2, (ADAP))
4. $\{S^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}} \wedge o = \mathsf{this}\}\ v := o.m(\overline{e})\ \{Q\}$ (3, (CONS))
5. $\{P \wedge o = \mathsf{this}\}\ v := o.m(\overline{e})\ \{Q\}$ (4, *def*)
6. $\{\forall v'.Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash i\vdash c_{(o,v')}} \wedge o \neq \mathsf{this}\}\ v := o.m(\overline{e})\ \{Q\}$ (CALLSYNC1)
7. $\{(P \wedge o = \mathsf{this}) \vee (\forall v'.Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash i\vdash c_{(o,v')}} \wedge o \neq \mathsf{this})\}\ v := o.m(\overline{e})\ \{Q\}$ (5, 6, (DISJ))
8. $\{\forall v' \,.\, \texttt{if}\ o = \mathsf{this}\ \texttt{then}\ P\ \texttt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash i\vdash c_{(o,v')}}\}\ v := o.m(\overline{e})\ \{Q\}$ (7, *math*)
9. $\{o = \mathsf{null}\}\ v := o.m(\overline{e})\ \{false\}$ (NOTNULL)
10. $\{o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \texttt{if}\ o = \mathsf{this}\ \texttt{then}\ P\ \texttt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash i\vdash c_{(o,v')}}\}$
    $v := o.m(\overline{e})\{Q\}$ (8, 9, (DISJ))
11. $\{wlp(v := o.m(\overline{e}), Q)\}\ v := o.m(\overline{e})\ \{Q\}$ (10, *def*)

Statement $o.m(\overline{e})$.
Following the same outline as for statement $v := o.m(\overline{e})$ above, $wlp(o.m(\overline{e}), Q)$ can be written as:

$$o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \texttt{if}\ o = \mathsf{this}\ \texttt{then}\ (wlp(m(\overline{x})body, Q^{\mathcal{H},\overline{y}}_{\mathcal{H}\vdash c_{(\mathsf{this},\mathsf{return})},\overline{y}'}))^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}}\ \texttt{else}\ Q^{\mathcal{H}}_{\mathcal{H}\vdash i\vdash c_{(o,v')}}$$

Below we let $P$ denote the assertion:

$$(wlp(m(\overline{x})body, Q^{\mathcal{H},\overline{y}}_{\mathcal{H}\vdash c_{(\mathsf{this},\mathsf{return})},\overline{y}'}))^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}}$$

Assertion $S$ denotes $wlp(m(\overline{x})body, Q^{\mathcal{H},\overline{y}}_{\mathcal{H}\vdash c_{(\mathsf{this},\mathsf{return})},\overline{y}'})$, and $R$ denotes $Q^{\mathcal{H},\overline{y}}_{\mathcal{H}\vdash c_{(\mathsf{this},\mathsf{return})},\overline{y}'}$. The proof then corresponds to the one above:

1. $\{S\}\ m(\overline{x})body\ \{R\}$ (*premise*)
2. $\{S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this}\}$
   $o.m(\overline{e})\{\exists v'\,.\, R^{\mathsf{return},\mathsf{caller},\mathcal{H}}_{v',\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v')}\}$ (1, (callSync2-1))
3. $\{\forall \overline{w}', \mathcal{H}'\,.\,(\forall \overline{y}', \overline{l}\,.\, S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this} \Rightarrow$
   $(\exists v'\,.\, R^{\mathsf{return},\mathsf{caller},\mathcal{H}}_{v',\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v')})^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}\}$
   $o.m(\overline{e})\{Q\}$ (2, (adap))
4. $\{S^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}} \wedge o = \mathsf{this}\}\ o.m(\overline{e})\ \{Q\}$ (3, (cons))
5. $\{P \wedge o = \mathsf{this}\}\ o.m(\overline{e})\ \{Q\}$ (4, *def*)
6. $\{\forall v'.Q^{\mathcal{H}}_{\mathcal{H}\ \vdash i\vdash c_{(o,v')}} \wedge o \neq \mathsf{this}\}\ o.m(\overline{e})\ \{Q\}$ (callSync1-1)
7. $\{(P \wedge o = \mathsf{this}) \vee (\forall v'.Q^{\mathcal{H}}_{\mathcal{H}\ \vdash i\vdash c_{(o,v')}} \wedge o \neq \mathsf{this})\}\ o.m(\overline{e})\ \{Q\}$ (5, 6, (disj))
8. $\{\forall v'\,.\,\mathtt{if}\ o = \mathsf{this}\ \mathtt{then}\ P\ \mathtt{else}\ Q^{\mathcal{H}}_{\mathcal{H}\ \vdash i\vdash c_{(o,v')}}\}\ o.m(\overline{e})\ \{Q\}$ (7, *math*)
9. $\{o = \mathsf{null}\}\ o.m(\overline{e})\ \{false\}$ (notNull)
10. $\{o \neq \mathsf{null} \Rightarrow \forall v'\,.\,\mathtt{if}\ o = \mathsf{this}\ \mathtt{then}\ P\ \mathtt{else}\ Q^{\mathcal{H}}_{\mathcal{H}\ \vdash i\vdash c_{(o,v')}}\}$
    $o.m(\overline{e})\{Q\}$ (8, 9, (disj))
11. $\{wlp(o.m(\overline{e}), Q)\}\ o.m(\overline{e})\ \{Q\}$ (10, *def*)

Rule (method).

The side condition $\overline{y} \notin FV[Q]$ means that $\exists \overline{y}\,.\,Q = Q$. By the rule premise, we have $S = wlp(s', wf(\mathcal{H}) \Rightarrow Q)$, where $s'$ is as for the soundness proof of (method) above. The remaining verification condition $wlp(\mathtt{var}\ \overline{y}, S) \Rightarrow \forall \overline{y}\,.\,S$ is trivial.

# A.8 Complete Code of Publisher-Subscriber Example

```
data News = E1 | E2 | E3 | E4 | E5 | None;

interface ServiceI{
        Void subscribe(ClientI cl);
        Void produce()}

interface ProxyI{
        ProxyI add(ClientI cl);
        Void publish(Fut<News> fut)}

interface ProducerI{
        News detectNews()}

interface NewsProducerI{
        Void add(News ns);
        News getNews();
        List<News> getRequests()}

interface ClientI{
        Void signal(News ns)}


class Service(Int limit, NewsProducerI np) implements ServiceI{
    ProducerI prod; ProxyI proxy; ProxyI lastProxy;
    {prod := new Producer(np); proxy := new Proxy(limit,this); lastProxy := proxy; this!produce()}

    Void subscribe(ClientI cl){lastProxy := lastProxy.add(cl)}

    Void produce(){var Fut<News> fut := prod!detectNews(); proxy!publish(fut)}}


class Proxy(Int limit, ServiceI s) implements ProxyI{
    List<ClientI> myClients := Nil; ProxyI nextProxy;

    ProxyI add(ClientI cl){
      var ProxyI lastProxy = this;
      if length(myClients) < limit then myClients := appendright(myClients, cl)
      else if nextProxy == null then nextProxy := new Proxy(limit,s) fi;
      lastProxy := nextProxy.add(cl) fi; put lastProxy}

    Void publish(Fut<News> fut){
      var News ns = None;
      ns = fut.get; myClients!signal(ns);
      if nextProxy == null then s!produce() else nextProxy!publish(fut) fi}}

class Producer(NewsProducerI np) implements ProducerI{
        News detectNews(){
          var List<News> requests := Nil; News news := None;
          requests := np.getRequests();
          while requests == Nil do requests := np.getRequests() od
          news := np.getNews(); put news}}

class NewsProducer implements NewsProducerI{
        List<News> requests := Nil;
        Void add(News ns){requests := appendright(requests,ns)}
        News getNews(){var News firstNews := head(requests); requests := tail(requests); put firstNews}
        List<News> getRequests(){put requests}}

class Client implements ClientI{
        News news := None;
        Void signal(News ns){news := ns}}
```

We have here augmented the given core language with ABS syntax for data types.

# Bibliography


[1] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theoretical Computer Science*, 331(2-3):251–290, Feb. 2005.

[2] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, 78(7):491–518, 2009.

[3] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(2):3 –14, may 1993.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1998.

[5] A. Ahern and N. Yoshida. Formalising Java RMI with explicit code mobility. *Theoretical Computer Science*, 389(3):341 – 410, 2007.

[6] W. Ahrendt and M. Dylla. A verification system for distributed objects with asynchronous method calls. In K. Breitman and A. Cavalcanti, editors, *Proc. International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *LNCS*, pages 387–406. Springer-Verlag, 2009.

[7] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 77(12):1289–1309, Oct. 2012.

[8] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[9] G. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 1999.

[10] K. R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.

[11] K. R. Apt. Ten years of Hoare's logic: A survey — Part II: Nondeterminism. *Theoretical Computer Science*, 28(1–2):83–109, Jan. 1984.

[12] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, Sept. 1986.

[13] J. Armstrong, R. Virding, and M. Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

[14] H. G. Baker Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[15] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3:2004, 2004.

[16] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[17] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[18] M. Broy and K. Stølen. *Specification and Development of Interactive Systems*. Monographs in Computer Science. Springer-Verlag, 2001.

[19] R. Bubel, R. Hähnle, and U. Geilmann. A formalisation of Java strings for program specification and verification. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, SEFM'11, pages 90–105, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *SIGPLAN Not.*, 27(8):15–42, Aug. 1992.

[21] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, May 2005.

[22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[23] O.-J. Dahl. Can program proving be made practical? In M. Amirchahy and D. Néel, editors, *Les Fondements de la Programmation*, pages 57–114. Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, Dec. 1977.

[24] O.-J. Dahl. Object-oriented specifications. In *Research directions in object-oriented programming*, pages 561–576. MIT Press, Cambridge, MA, USA, 1987.

[25] O.-J. Dahl. *Verifiable Programming*. International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.

[26] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. (Simula 67) Common Base Language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.

[27] O.-J. Dahl and O. Owe. Formal methods and the RM-ODP. Technical Report Research Report 261, Dept. of Informatics, Univ. of Oslo, 1998. Full version of a paper presented at NWPT'98: Nordic Workshop on Programming Theory, Turku.

[28] F. S. de Boer. A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. *Theoretical Computer Science*, 274:3–41, 2002.

[29] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, Mar. 2007.

[30] F. S. de Boer and C. Pierik. How to cook a complete Hoare logic for your pet OO language. In *Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 111–133. Springer-Verlag, 2004.

[31] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency verification: introduction to compositional and non-compositional methods*. Cambridge University Press, New York, NY, USA, 2001.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[33] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[34] C. C. Din, R. Bubel, and O. Owe. A comparison of runtime assertion checking and theorem proving for concurrent and distributed systems (extended abstract). In *Proceeding of the Nordic Workshop on Programming Theory (NWPT'13)*, 2013.

[35] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. Research Report 401, Nov. 2010. Available from `https://www.duo.uio.no/handle/10852/8806`.

[36] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects (extended abstract). In *Proceeding of the Nordic Workshop on Programming Theory (NWPT'10)*, 2010.

[37] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.

[38] C. C. Din, J. Dovland, and O. Owe. An approach to compositional reasoning about concurrent objects and futures. Research Report 415, Dept. of Informatics, University of Oslo, Feb. 2012. Available from `https://www.duo.uio.no/handle/10852/9050`.

[39] C. C. Din, J. Dovland, and O. Owe. Compositional reasoning about shared futures. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc. International Conference on Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of *LNCS*, pages 94–108. Springer-Verlag, 2012.

[40] C. C. Din, J. Dovland, and O. Owe. Soundness of a reasoning system for asynchronous communication with futures (extended abstract). In *Proceeding of the Nordic Workshop on Programming Theory (NWPT'12)*, 2012.

[41] C. C. Din, J. Dovland, and O. Owe. A comparison of runtime assertion checking and theorem proving for concurrent and distributed systems. Research Report 435, Dept. of Informatics, University of Oslo, Nov. 2013. Available at `https://www.duo.uio.no/handle/10852/38331`.

[42] C. C. Din and O. Owe. Compositional and sound reasoning about active objects with shared futures. Research Report 437, Dept. of Informatics, University of Oslo, Feb. 2014. Available at `https://www.duo.uio.no/handle/10852/38332`.

[43] C. C. Din, O. Owe, and R. Bubel. Runtime assertion checking and theorem proving for concurrent and distributed systems. In *Proceedings of the 2.nd Intl. Conf. on Model-Driven Engineering and Software Development,* Modelsward'14, pages 480–487. SCITEPRESS, 2014. DOI: 10.5220/0004877804800487.

[44] J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *Proceedings of the IEEE International Conference on Software Science, Technology & Engineering (SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.

[45] J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. In D. Goldin and F. Arbab, editors, *Proc. Workshop on the Foundations of Interactive Computation (FInCo'07)*, volume 203 of *Electr. Notes Theor. Comput. Sci.*, pages 19–34. Elsevier, May 2008.

[46] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

[47] K. E. K. Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing asynchronous remote method invocation in Java. In *6th Australian Conference on Parallel and Real-Time Systems*, pages 22–34. Springer-Verlag, 1999.

[48] X. Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM.

[49] G. Gentzen. Untersuchungen über das logische Schließen I,II. *Mathematische Zeitschrift*, 1934.

[50] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[51] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[52] R. Hähnle. The abstract behavioral specification language: A tutorial introduction. In E. Giachino, R. Hähnle, F. Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer Berlin Heidelberg, 2013.

[53] R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[54] P. B. Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34:38–45, 1999.

[55] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.

[56] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

[57] Full ABS Modeling Framework (Mar 2011). Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[58] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *The C♯ Programming Language*. Addison-Wesley Professional, 3rd edition, 2008.

[59] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[60] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[61] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[62] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Oct. 1974.

[63] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[64] J. D. Ichbiah, B. Krieg-Brueckner, B. A. Wichmann, J. G. P. Barnes, O. Roubine, and J.-C. Heliard. Rationale for the design of the Ada programming language. *SIGPLAN Not.*, 14(6b):1–261, June 1979.

[65] International Telecommunication Union. Open Distributed Processing – Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[66] A. S. A. Jeffrey and J. Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *Proc. European Symposium on Programming*, volume 3444 of *LNCS*, pages 423–438. Springer-Verlag, 2005.

[67] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[68] E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 137–164. Springer-Verlag, 2004.

[69] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[70] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, Oct. 1983.

[71] T. Kleymann. Hoare logic and auxiliary variables. *Formal Asp. Comput.*, 11(5):541–566, 1999.

[72] J. L. Knudsen. Name collision in multiple classification hierarchies. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '88, pages 93–109, London, UK, UK, 1988. Springer-Verlag.

[73] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Research directions in object-oriented programming. chapter The BETA Programming Language, pages 7–48. MIT Press, Cambridge, MA, USA, 1987.

[74] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual, 2008.

[75] K. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer Berlin Heidelberg, 2009.

[76] K. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer Berlin Heidelberg, 2006.

[77] N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[78] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM.

[79] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.

[80] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[81] B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[82] B. Morandi, S. S. Bauer, and B. Meyer. SCOOP – a contract-based concurrent object-oriented programming model. In P. Müller, editor, *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 41–90. Springer, 2008.

[83] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992.

[84] M. Nordio, C. Calcagno, P. Müller, and B. Meyer. Soundness and completeness of a program logic for Eiffel. Technical Report 617, ETH Zurich, 2009.

[85] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[86] E.-R. Olderog and K. R. Apt. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages*, 10(3):420–455, July 1988.

[87] O. Owe. Axiomatic treatment of processes with shared variables revisited. *Formal Asp. Comput.*, 4(4):323–340, 1992.

[88] C. Pierik and F. Boer. A syntax-directed Hoare logic for object-oriented programming concepts. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 64–78. Springer Berlin Heidelberg, 2003.

[89] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[90] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *FOCS*, pages 109–121, 1976.

[91] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[92] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In T. D'Hondt, editor, *(ECOOP 2010)*, volume 6183 of *LNCS*, pages 275–299. Springer-Verlag, June 2010.

[93] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, Oct. 1984.

[94] N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31(1-2):13–29, May 1984.

[95] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[96] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. Vasconcelos, editors, *CONCUR 2007 Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin Heidelberg, 2007.

[97] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, Nov. 1986.